intel® *Development Solutions*

PL/M Programmer's Guide

PL/M Programmer's Guide

Order Number: 452161-001

PL/M PROGRAMMER'S GUIDE

Order Number: 452161-001

REV.	REVISION HISTORY	DATE
-001	Original Issue.	10/87

Above	im	iSBC	PC BUBBLE
BITBUS	iMDDX	iSBX	Plug-A-Bubble
COMMputer	iMMX	iSDM	PROMPT
CREDIT	Insite	iSXM	Promware
Data Pipeline	Intel	KEPROM	QueX
FASTPATH	int$_e$l	Library Manager	QUEST
GENIUS	int$_e$lBOS	MAP-NET	Quick-Pulse Programming
$\overset{\Delta}{i}$	Intelevision	MCS	Ripplemode
i	int$_e$ligent Identifier	Megachassis	RMX/80
I^{2}ICE	int$_e$ligent Programming	MICROMAINFRAME	RUPI
ICE	Intellec	MULTIBUS	Seamless
iCEL	Intellink	MULTICHANNEL	SLD
iCS	iOSP	MULTIMODULE	UPI
iDBP	iPDS	ONCE	VLSiCEL
iDIS	iPSC	OpenNET	4-SITE
iLBX	iRMX	OTP	

Contents

Tabs for
452161-001

CONTENTS

CHAPTER 8 PROCEDURES

CHAPTER 9 BUILT-IN PROCEDURES, FUNCTIONS, AND VARIABLES

TABLES

PREFACE

This manual, the *PL/M Programmer's Guide*, describes the PL/M-86, PL/M-286 and PL/M-386 languages and gives instructions for using the corresponding PL/M compilers. The *PL/M Programmer's Guide* assumes basic familiarity with programming concepts, including structured programming. It is organized as follows:

- Chapter 1 is an overview of the PL/M language. It includes an introductory sample program.
- Chapter 2 defines the basic elements of the PL/M language.
- Chapter 3 concerns types, declarations, variables, and related topics.
- Chapter 4 summarizes the rules for declaring and referencing arrays and structures.
- Chapter 5 describes expressions and assignment statements.
- Chapter 6 concerns the statements that control the flow of program execution.
- Chapter 7 discusses block structure and scope.
- Chapter 8 describes procedure declaration, activation, and attributes.
- Chapter 9 concerns built-in procedures, functions, and variables.
- Chapter 10 describes the built-in functions that manipulate the target microprocessor and numeric coprocessor.
- Chapter 11 explains each of the compiler controls and their effect on compiler output.
- Chapter 12 presents an annotated sample PL/M-86 program.
- Chapter 13 discusses the extended segmentation models and controls.
- Chapter 14 lists the error and warning messages and provides brief explanations of the messages.
- Appendix A lists reserved words and predeclared identifiers.
- Appendix B summarizes program limits.
- Appendix C is a formal summary of PL/M syntax.
- Appendix D summarizes the differences among various dialects of PL/M.
- Appendix E is an ASCII-character table that also identifies which are PL/M characters.

- Appendix F explains how to link to modules written in other languages.
- Appendix G discusses run-time interrupt processing.
- Appendix H summarizes the libraries supplied with PL/M.

Note that throughout this manual, the general term PL/M refers to PL/M-86, PL/M-286, and PL/M-386. Information that is peculiar to a particular PL/M compiler is flagged with the notation shown below.

PL/M-###

end PL/M-###

Related Publications

To use a PL/M-86, PL/M-286, or PL/M-386 compiler on any supported host, you need the *PL/M Programmer's Guide* (this manual) and one of the following host-oriented supplements.

DOS Supplement for the PL/M Programmer's Guide, order number 452162

VAX/VMS Supplement for the PL/M Programmer's Guide, order number 480609

XENIX Supplement for the PL/M Programmer's Guide, order number 452165

iRMX® Supplement for the PL/M Programmer's Guide, order number 452164

ISIS/iNDX Supplement for the PL/M Programmer's Guide, order number 452163

Introduction 1

Tabs for
452161-001

1 INTRODUCTION CONTENTS

1 INTRODUCTION

This chapter introduces the PL/M-86, PL/M-286, and PL/M-386 languages and explains the process of developing software for execution by an 8086/80186-, 80286-, or 80386-based system.

1.1 Product Definition

PL/M is a high-level language for programming Intel microprocessors. It was designed by Intel Corporation to meet the software requirements of computers in a wide variety of systems and applications work.

The PL/M compilers are software tools that translate PL/M source code into relocatable object modules. These modules can then be combined with other modules coded in PL/M, assembly language, or other high-level languages. The compilers provide listing output, error messages, and a number of compiler controls that aid in developing and debugging programs.

You can use LINK86, BND286, or BND386 to combine the program modules, and locate the programs with LOC86 or the 80286 or 80386 System Builder. You can then combine the resulting relocatable object modules with the necessary support libraries.

After the program modules are combined and located, you can use an in-circuit emulator (e.g., the ICE™-386 emulator) or a software debugging utility (e.g., PSCOPE) to debug the program.

For firmware systems, to transfer the program to PROM, use the Universal PROM Programmer (iUPP) and the Universal PROM Mapper (UPM) software.

1.2 Advantages of Using the PL/M Language

PL/M programs are portable, which means that they are easily transferred from one microprocessor to another. When using PL/M, you need not be concerned with the instruction set of the target processor. Additionally, there is no need to be concerned with other details of the target processor, such as register allocation or assigning the proper number of bytes for each data item. The PL/M compilers do these functions automatically. PL/M keywords and phrases are close to natural English, and many operations (including arithmetic and Boolean operations) can be combined into expressions. This enables the execution of a sequence of operations with just one program statement. Data types and data structures are close to the actual problem. For instance, in PL/M, the program can be written in terms of Boolean expressions, characters, and data structures, in addition to bytes, words, and integers. The introductory example at the end of this chapter illustrates these points.

Coding programs in a high-level language rather than assembly language involves thinking closer to the level used when planning the overall system design. Following is a list of the advantages of using PL/M, and the applications for which PL/M is best suited:

- PL/M block structure and control constructs aid and encourage structured programming.
- PL/M has facilities for data structures such as structured arrays and pointer-based dynamic variables.
- PL/M is a typed language. The compiler does data type compatibility checking during compilation to help detect logic errors in programs.
- PL/M data structuring facilities and control statements are designed in a logically consistent way. Thus, PL/M is a good language for expressing algorithms for systems programming.
- PL/M is a standard language used on Intel microcomputers. PL/M programs are upwardly compatible across the 80[x]86 family of microprocessors.
- PL/M was designed for programmers (generally systems programmers) who need access to the microprocessor's features such as indirect addressing and direct I/O for optimum use of all system resources.

In comparison with other languages, PL/M has more features than BASIC and is a more general-purpose language than either FORTRAN (best suited for scientific applications) or COBOL (designed for business data processing). PL/M accesses the microprocessor hardware features more easily than C. Additionally, in comparison to C, PL/M offers the ability to nest procedures and the program structure is easier to maintain.

1.3 The Structure of a PL/M Program

PL/M is a block-structured language; every statement in a program is part of at least one block. A block is a well-defined group of statements that begins with a DO statement or a procedure declaration and ends with an END statement.

A module is a labeled simple DO-block. A module must begin with a labeled DO statement and end with an END statement. Between the DO statement and the END statement other statements provide the definitions of data and processes that make up the program. These statements are said to be part of the block, contained within the block, or nested within the block. A module can contain other blocks but is never itself contained within another block. See Chapter 6 for a description of DO-blocks.

Every PL/M program consists of one or more modules, separately compiled, each consisting of one or more blocks. The two kinds of blocks are DO-blocks and procedure definition blocks.

A procedure definition block is a set of statements beginning with a procedure declaration and ending with an END statement. Other declarations and executable statements can be placed between these points, and are used later when the procedure is actually invoked or called into execution. The definition block is a further declaration of everything the procedure will use and do.

1.4 Overview of PL/M Statements

The two types of statements in PL/M are declarations and executable statements. All PL/M statements end with a semicolon (;).

1.4.1 Declaration Statements

The following is a simple example of a declaration statement:

```
DECLARE WIDTH BYTE;
```

This statement introduces the identifier WIDTH and associates it with the contents of 1 byte (8 bits) of memory. Now, rather than having to know the memory address of this byte, you can refer to it by the name WIDTH.

A group of statements intended to perform a function (i.e., a subprogram or subroutine) can be given a name by declaring them to be a procedure:

```
ADDER_UPPER: PROCEDURE (BETA) BYTE;
```

The statements that define the procedure follow the semicolon. This block of PL/M statements is invoked from other points in the program, and may involve passing parameters to the program. When a procedure has finished executing, control is returned immediately to the main program. This capability is the major feature enabling modular program construction.

1.4.2 Executable Statements

The following is an example of an executable statement:

```
CLEARANCE = WIDTH + 2;
```

The two identifiers, CLEARANCE and WIDTH, must be declared prior to this executable statement, which produces machine code to retrieve the WIDTH value from memory. Once the WIDTH value is obtained, 2 is added to it and the sum is stored in the memory location for CLEARANCE.

For most purposes, it is unnecessary to think in terms of memory locations when programming in PL/M. CLEARANCE and WIDTH are variables, and the assignment statement assigns the value of the expression WIDTH + 2 to the variable CLEARANCE. The compiler automatically generates all the machine code necessary to retrieve data from memory, to evaluate the expression retrieved, and to store the result in the proper location.

Executable statements are discussed in the following chapters:

Assignment Statement	Chapter 5
CALL Statement	Chapter 8
CAUSE$INTERRUPT Statement	Chapter 10
DISABLE Statement	Chapter 10
DO CASE Statement	Chapter 6
DO WHILE Statement	Chapter 6
ENABLE Statement	Chapter 10
END Statement	Chapter 6
Executable Functions	Chapter 9
GOTO Statement	Chapter 6
HALT Statement	Chapter 10
IF Statement	Chapter 6
Iterative DO Statement	Chapter 6
Nested IF Statement	Chapter 6
RETURN Statement	Chapter 8
Simple DO Statement	Chapter 6

1.4.3 Built-In Procedures and Variables

PL/M provides a variety of built-in procedures and variables. These include functions such as shifts and rotations, data type conversions, executable functions, block I/O, real math, and string manipulation (see Chapters 9 and 10 for details).

1.4.4 Overview of PL/M Expressions

A PL/M expression is made up of operands and operators, and resembles a conventional algebraic expression.

Operands include numeric constants (such as 3.78 or 105) and variables (as well as other types discussed in Chapters 3 and 5). The operators include + and − for addition and subtraction, * and / for multiplication and division, and MOD for modulo arithmetic.

As in an algebraic expression, elements of a PL/M expression can be grouped with parentheses.

An expression is evaluated using unsigned binary arithmetic, signed integer arithmetic, and/or floating-point arithmetic, depending on the types of operands in the expression (see Chapters 3 and 5).

1.4.5 Input and Output

PL/M does not provide formatted I/O capabilities like those of FORTRAN, BASIC, or COBOL. However, PL/M does provide built-in functions for direct I/O that do not require operating system run-time support. In PL/M-86, these built-in functions allow for single-byte or single-word I/O. In PL/M-286 (and in PL/M-86 programs compiled for execution on an 80186-based system), these built-in functions allow for single-byte or word I/O, and block I/O (for strings of bytes or single words). In PL/M-386, these built-in functions allow for single-byte, half-word or word I/O, as well as for block I/O (for strings of bytes, half-words, or single-words). For detailed information on these I/O functions, see Chapter 10.

1.5 An Introductory Sample Program

Figure 1-1 shows a sample PL/M-86 program. This program contains many undefined words and constructs that will be explained in the upcoming chapters.

The main program does little but define data and call the procedure named SORTPROC. To prepare this sample program for execution, type the program using a text editor.

To compile the module, if the file PROG1A.SRC contains the module SORTPROC and the compiler and your source file are in the current directory, invoke the compiler with the following command:

```
PLM86 PROG1A.SRC<cr>
```

Host systems vary in their invocation procedures. The compiler responds with the following sign-on message:

```
system-id PL/M-86 COMPILER Vx.y
```

Where:

system-id	is the host operating system name.
x.y	is the compiler version number.

This is normally followed by the console sign-off message:

```
PL/M-86 COMPILATION COMPLETE. n WARNING[S], m ERROR[S]
```

Where:

n and *m*	represent the number of warning and nonfatal error messages generated during compilation.

NOTE

To generate and output sort data, user-supplied routines must be added to this program in the areas indicated.

For example, interfacing with the operating system can be done using UDI calls (e.g., DQ$WRITE, DQ$EXIT).

Interface routines must be used to exit to the system and return control to the host operating system (e.g., DQ$EXIT in UDI). Each operating environment has its own set of interface routines.

```
PL/M-86 COMPILER   SORTMODULE                         mm/dd/yy hh:mm:ss   PAGE 1

system-id PL/M-86 Vx.y  COMPILATION OF MODULE SORTMODULE
OBJECT MODULE PLACED IN prog1a.obj
COMPILER INVOKED BY: path PLM86.86 prog1a.src PW(80)

  1        SORTMODULE: DO;                                  /* Beginning of module */

  2  1     MOVBYTES: PROCEDURE (SRC$PTR, DEST$PTR, SIZE);
  3  2       DECLARE (SRC$PTR,DEST$PTR) POINTER,
                      SIZE WORD;

  4  2       IF (SRC$PTR > DEST$PTR) THEN
  5  2          CALL MOVB(SRC$PTR,DEST$PTR,SIZE);
  6  2       ELSE
                CALL MOVRB (SRC$PTR,DEST$PTR,SIZE);
  7  2     END MOVBYTES;

  8  1     SORTPROC: PROCEDURE (PTR, COUNT, RSIZE, KEY);
  9  2       DECLARE PTR POINTER, (COUNT, RSIZE, KEY) WORD;

           /* Parameters:
               PTR is pointer to first record.
               COUNT is number of records to be sorted.
               RSIZE is number of bytes in each record,
                 max is 128.
               KEY is byte position within each record of a BYTE
                 scalar to be used as sort key. */

 10  2        DECLARE REC BASED PTR(1) BYTE,
                CUR (128) BYTE,
                (I, J) WORD;

 11  2        SORT:
                DO J = 1 TO COUNT - 1;
 12  3           CALL MOVBYTES(@REC(J*RSIZE), @CUR, RSIZE);
 13  3           I = J;
 14  3           DO WHILE I > 0
                    AND (REC (*RSIZE+KEY) > CUR(KEY) );
 15  4             CALL MOVBYTES(@REC( (I-1)*RSIZE),
                      @REC(I*RSIZE), RSIZE);
```

Figure 1-1 Sample Program: Module Sort

```
PL/M-86 COMPILER   SORTMODULE                         mm/dd/yy hh:mm:ss   PAGE 2

 16  4            I = I -1;
 17  4          END;
 18  3          CALL MOVBYTES(@CUR, @REC(I*RSIZE), RSIZE);
 19  3      END SORT
 20  2    END SORTPROC;
                                   /* Program to sort two sets of records, using SORTPROC.*/

 21  1    DECLARE SET1(50) STRUCTURE (
            ALPHA WORD,
            BETA(12) WORD,
            GAMMA INTEGER,
            DELTA REAL,
            EPSILON BYTE);

 22  1    DECLARE SET2(500) STRUCTURE (
            ITEMS(21) INTEGER
            VOLTS REAL,
            KEY BYTE);

                                       /* Data is read in to initialize the records. */

 23  1    CALL SORTPROC(@SET1, LENGTH(SET1), SIZE(SET1(0)),
                         SIZE(SET1(0).ALPHA));

 24  1    CALL SORTPROC(@SET2, LENGTH(SET2), SIZE(SET2(0))
                         (SIZE(SET2(0))-SIZE(SET2(0).KEY)));

 25  1    END SORTMODULE;                                         /* End of module */
```

Figure 1-1 Sample Program: Module Sort (continued)

```
PL/M-86 COMPILER    SORTMODULE                        mm/dd/yy hh:mm:ss   PAGE 3

  MODULE INFORMATION:

      CODE AREA SIZE     = 00FBH     251D
      CONSTANT AREA SIZE = 0000H       0D
      VARIABLE AREA SIZE = 62C2H   25282D
      MAXIMUM STACK SIZE = 0018H      24D
      64 LINES READ
      0 PROGRAM WARNINGS
      0 PROGRAM ERRORS

  DICTIONARY SUMMARY:

     471KB MEMORY AVAILABLE
     4KB DISK SPACE USED
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

Figure 1-1 Sample Program: Module Sort (continued)

Language Elements 2

Tabs for
452161-001

2 LANGUAGE ELEMENTS CONTENTS

2 LANGUAGE ELEMENTS

PL/M-86, -286 and -386 programs are "free-form", meaning that there are no restrictions on where you place a statement on a line. You can use as many blanks (spaces) as necessary to format your program for readability.

2.1 Character Set

The PL/M-86/286/386 source program character set is the following subset of the ASCII character set:

A . . Z
a . . z
0 . . 9

and the following special characters:

= . / ( ) + – ' * , < > : ; @ $ _

and the blank (space), tab, carriage-return and line-feed characters. (Appendix E indicates whether each ASCII character is a member of the PL/M character set, and gives its hexadecimal value.)

PL/M does not distinguish between uppercase and lowercase letters, except in string constants. For example, the variable names "xyz" and "XYZ" are the same. (In this manual, all PL/M syntax is uppercase, by convention.)

Special characters have particular meaning in PL/M, as explained throughout this manual. Table 2-1 summarizes the meaning of special characters in PL/M.

The PL/M compilers treat multiple contiguous blanks in PL/M source programs as single blanks, by ignoring all the blanks except the first one.

The PL/M compiler produce an error or warning message whenever they encounter a character other than those described above in a source program

In addition to the source character set, PL/M allows the use of special character sets (such as Kanji characters), located from 0080H through 00FFH (excluding 0081H).

Table 2-1 PL/M Special Characters

Symbol	Name	Use
=	equal sign	Two distinct uses: (1) assignment operator (2) relational test operator
:=	assign	embedded assignment operator
@	at-sign	location reference operator
.	dot	Three distinct uses: (1) decimal point (2) structure member qualification (3) address operator
/	slash	division operator
/*		beginning-of-comment delimiter
*/		end-of-comment delimiter
(	left paren	left delimiter of lists, subscripts, and some expressions
)	right paren	right delimiter of lists, subscripts, and some expressions
+	plus	addition or unary plus operator
–	minus	subtraction or unary minus operator
'	apostrophe	string delimiter
*	asterisk	Two distinct uses: (1) multiplication operator (2) implicit dimension specifier
<	less than	relational test operator
>	greater than	relational test operator
<=	less or equal	relational test operator
>=	greater or equal	relational test operator
<>	not equal	relational test operator
:	colon	label terminator
;	semicolon	statement terminator
,	comma	list element delimiter
_	underscore	significant character in identifier
$	dollar sign	Two distinct uses: (1) non-significant character embedded within number or identifier (2) significant as the first character on a control line in a source file

2.2 Tokens, Separators, and the Use of Blanks

The smallest meaningful unit of a PL/M statement is a token. Every token belongs to one of the following classes:

- Identifiers
- Reserved words
- Simple delimiters (all of the special characters, except the dollar sign, are simple delimiters)
- Compound delimiters (combinations of two special characters):

  <>, <=, >=, :=, /*, */

- Numeric constants
- Character string constants

It is usually clear where one token ends and the next one begins. For example, in the assignment statement:

```
EXACT=APPROX*(HEIGHT-3)/SCALE;
```

EXACT, APPROX, HEIGHT, and SCALE are identifiers, 3 is a numeric constant, and all the other characters are simple delimiters.

If a delimiter (simple or compound) does not naturally occur between two tokens, you must separate them with one or more blank(s).

A comment can also be used as a separator.

Blanks can be inserted around any token without changing the meaning of the PL/M statement. Thus, the assignment statement:

```
EXACT = APPROX * ( HEIGHT - 3 ) / SCALE;
```

is equivalent to:

```
EXACT=APPROX*(HEIGHT-3)/SCALE;
```

2.3 Identifiers and Reserved Words

Identifiers name variables, procedures, symbolic constants, and statements. Statement identifiers are called labels. Identifiers can be up to 31 characters long. The first character must be alphabetic or the underscore (_), and the remaining characters may be alphabetic, numeric, or the underscore.

You can use the dollar sign character to improve the readability of an identifier or constant, but the dollar character is not meaningful to the PL/M compilers. An identifier or constant containing a dollar sign is equivalent to the same identifier without the dollar sign. Note that you must not use a dollar character in a procedure name within a subsystem definition. (See Chapter 13.)

Examples of valid identifiers are:

```
INPUT_COUNT
X
GAMM
LONGIDENTIFIERNUMBER3
LONG$$$IDENTIFIER$$$NUMBER$$$3
_MAIN
INPUT$COUNT
INPUTCOUNT
```

The long identifiers are identical to the compiler. INPUT$COUNT and INPUTCOUNT are interchangeable, but are different from INPUT_COUNT.

Identifiers must be distinct from reserved words. If you want to use PL/M built-in procedures and variables, the identifiers in your source program must be distinct from the built-ins' predefined identifiers. Appendix A lists the reserved words and predefined identifiers.

2.4 Constants

A constant is a value that does not change during a program's execution. The three types of constants are whole-number constants, floating-point constants, and character strings.

2.4.1 Whole-Number Constants

Whole-number constants can be binary, octal, decimal, or hexadecimal numbers. The PL/M compilers interpret numbers without a base suffix as decimal numbers. When they encounter characters that are invalid in the specified (or assumed) base, the compilers produce appropriate messages. If a constant contains characters invalid in the designated number base, it will be flagged as an error.

A whole-number constant can be an 8-bit, 16-bit or 32-bit value. In PL/M-386, a whole-number constant can also be a 64-bit value. The range of whole-number constants is non-negative. (The minus sign in front of a whole-number constant is not part of the constant.

The first character of a hexadecimal number must be a numeric digit to avoid looking like an identifier. For example, write the hexadecimal form of the decimal value 163 as "0A3H" (rather than "A3H"); otherwise the compilers will interpret it as an identifier.

Examples of valid whole-number constants are:

```
12AH 2 33Q 1010B 55D 0BF3H 65535 777O 3EACH 0F76C05H
```

Examples of invalid whole-number constants are:

`12AF`	Hexadecimal digits used without an H suffix, and invalid in the default decimal interpretation.
`12AD`	The final D could be a suffix but the A is not a decimal digit. If hexadecimal is intended, a final H is needed.
`11A2B`	A and 2 are not valid binary digits. If hexadecimal is intended, a final H is necessary.
`2ADGH`	G is not a valid hexadecimal digit.

For example, the maximum whole-number 16-bit constant is:

```
2**16-1 = 1111$1111$1111$1111B = 177777Q = 65535D = 0FFFFH
```

PL/M-386

In the PL/M-386, the maximum whole-number 32-bit constant is:

```
2**32-1 =1111$1111$1111$1111$1111$1111$1111$1111B
        =37777777777Q
        =4294967295D
        =0FFFFFFFFH
```

end PL/M-386

2.4.2 Floating-Point Constants

The presence of a decimal point in a decimal constant creates a floating-point constant. Floating-point constants are represented in REAL precision (see Section 3.4). Only decimal real constants are allowed.

At least one decimal digit (e.g., 0) must precede the decimal point. A fractional part is optional after the decimal point, as is the base-ten exponent, which is indicated by the letter E. This exponent must have at least one digit. Note that no fractional exponents are possible.

In PL/M-86 and PL/M-286, the range for floating-point constants is 3.37 x 10**38 to 1.17 x 10**(− 38).

In PL/M-386, the range is − 2**(+ 128) to − 2**(− 126), zero, + 2**(− 126) to + 2**(+ 128). This range is approximately − 3.4 x 10**38 to − 8.4 x 10**(− 37), zero, and 8.4 x 10**(− 37) to 3.4 x 10**38.

The following are examples of valid floating-point constants:

```
5.30      176.0     1.88      3.14159      16.      222.2
53.0E-1   1.760E2   0.188E1   314159.E-5   1.6E+1   2.222E+2
```

Note that plus signs do not change the meaning of exponents.

The following are examples of invalid floating-point constants:

`6`	No decimal point
`1.3AH`	Hexadecimal not allowed in floating-point constants
`10.011B`	Binary not allowed
`7.52Q`	Octal not allowed
`4.8E1AH/2`	Only decimal constants in exponents; no hexadecimal, no expressions, no fractions

2.4.3 Character Strings

Character strings are printable ASCII characters enclosed within apostrophes. There are two types of character strings: string constants and character constants. A string constant is used to initialize variables or to pass a pointer. The maximum length of a string constant is 255. A character constant is used in expressions, and its value should fit into a double or machine word (32 bits). A string used as a character constant can contain from one to four characters.

To include an apostrophe in a string, write it as two apostrophes (e.g., the string '''Q' comprises 2 characters, an apostrophe followed by a Q). Values 0080H through 00FFH (excluding 0081H) can be used in a quoted character string. Spaces are allowed but line-feeds are not. The compiler represents character strings in memory as ASCII codes, one 7-bit character code to each 8-bit byte, with a high-order zero bit. Strings of length 1 translate to single-byte values. Character constants of length 2 translate to 16-bit values, and those of length 3 or 4 translate to 32-bit values. For example:

'A' is equivalent to 41H
'AG' is equivalent to 4147H
'AGR' is equivalent to 414752H
'AGRX' is equivalent to 41475258H

(See Appendix E, ASCII Characters, Hex Values, and PL/M Character Set.)

Therefore, character constants can be used as 8-bit, 16-bit, or 32-bit values. Character constants longer than 4 characters exceed the 32-bit capacity.

2.5 Comments

In PL/M, a comment is a sequence of characters delimited on the left by the character pair /* and on the right by the character pair */. These delimiters instruct the compiler to ignore any text between them and to consider such text as not part of the program.

A comment can contain any printable ASCII or special character and can also include space, carriage-return, line-feed, and tab characters. If you embed a comment in a character string constant, it becomes part of the constant. A comment can appear anywhere that a blank character can appear except embedded within a token.

The following is an example of a PL/M comment:

```
/*This procedure copies one structure to another.*/
```

In this manual, comments are presented in lowercase to distinguish them visually from program code, which is presented in uppercase.

Data Declarations, Types, and Based Variables

3

Tabs for
452161-001

3 DATA DECLARATIONS, TYPES, AND BASED VARIABLES CONTENTS

3 DATA DECLARATIONS, TYPES, AND BASED VARIABLES

In PL/M, you can declare symbolic names for variables, constants, procedures and statements ("labels"). For each symbolic name, there must be one declaration at the beginning of the block containing the name, or in an outer, enclosing block. A declaration consists of an identifier, type, attributes and/or location. Multiple declarations of a name in a block are invalid.

The declaration of a variable or constant identifier must precede use of the identifier in an executable statement. Although it is not good programming practice, you can call a reentrant procedure before defining it. You can either explicitly declare a statement label, or implicitly declare it by attaching it to an executable statement with a colon character.

Required and optional declaration elements are shown in Table 3-1.

Table 3-1 Declaration Elements

Declaration Statements For	Must Use	Can Use
Variable Names	BYTE, WORD, DWORD, INTEGER, POINTER, SELECTOR, REAL, STRUCTURE, ADDRESS* Additionally, for PL/M-386: HWORD, QWORD, CHARINT, SHORTINT, LONGINT, OFFSET	linkage attributes:** PUBLIC or EXTERNAL or location attributes: AT (location reference) variable initialization attribute: INITIAL (value-list)
Constant Names	type, as above, and constant initialization attribute: DATA (value-list)	linkage attributes as above
Label Names	LABEL	linkage attributes as above
Macro Substitution Names	LITERALLY 'string'	

*ADDRESS is equivalent to the OFFSET data type.
**Placement is important (see Section 3.1.1).

3.1 Variable Declaration Statements

A DECLARE statement is a nonexecutable statement that introduces some object or collection of objects, associates names (and sometimes values) with them, and allocates storage if necessary. The most important use of DECLARE is for declaring variables.

A variable can be a scalar (i.e., a single quantity), an array, or a structure.

A scalar variable is a single object whose value may not be known at compile time and may change during the execution of the program.

An array is a list of scalars with the same identifier, the individual objects of which are differentiated by subscripts.

A structure is an aggregate of scalars, arrays and/or structures with the same main identifier. The members of a structure are differentiated from each other by their own member-identifiers or field names. For example, EMPLOYEES.NAME would refer to the NAME field within the structure EMPLOYEES.

3.1.1 Sample DECLARE Statements

Note that when using linkage (PUBLIC/EXTERNAL) and initialization (DATA/INITIAL) attributes, the order of declaration is critical. Place linkage attributes before the initialization attribute, and after the type declaration.

For example:

```
DECLARE a$p BYTE PUBLIC INITIAL(4);
```

The following statements declare scalars:

```
    DECLARE APPROX REAL;
    DECLARE (OLD, NEW) BYTE;
    DECLARE POINT WORD, VAL12 BYTE;
```

The first example declares a single scalar variable of type REAL, with the identifier APPROX.

The second example declares two scalars, OLD and NEW, both of type BYTE. This kind of statement is called a factored declaration, which is similar to the sequence:

```
    DECLARE OLD BYTE;
    DECLARE NEW BYTE;
```

A factored declaration (for structures and arrays) guarantees that the bytes will be contiguously located in memory, which may be useful in real time applications (see also Section 3.1.3). Separate declaration statements do not guarantee this.

The third example declares two scalars of different types: POINT is of type WORD, and VAL12 is of type BYTE.

The following statements declare arrays:

```
DECLARE DOMAIN (128) BYTE;
DECLARE GAMMA (19) DWORD;
```

The first example declares the array DOMAIN, with 128 scalar elements of type BYTE. These elements are distinguishable by subscripting the name DOMAIN, using the range 0 to 127 for the subscripts. For example, the third element of DOMAIN can be referred to as DOMAIN(2). The first element of every array has subscript 0.

The second example declares the array GAMMA, with 19 scalar elements of type DWORD. The subscripts for this array can range from 0 to 18.

The third example declares a structure with two scalar members:

```
DECLARE RECORD STRUCTURE (KEY BYTE, INFO WORD);
```

The two members are a BYTE scalar that can be referred to as RECORD.KEY and a WORD scalar that can be referred to as RECORD.INFO. The word named by RECORD.INFO is the second and third bytes of this structure.

Structures are discussed in further detail in Chapter 4.

3.1.2 Results of Variable Declarations

Valid variable declarations result in the following:

- The name is given a unique address.
- The variable is considered to have the attributes declared.

All subsequent uses of the variable in the block where it is declared refer to the same address (except for based variables, discussed in Section 3.5).

A valid variable declaration also requires all references to the variable to conform to the rules for the current attributes (i.e., those attributes having priority in the current block). Thus, the compiler can flag a large variety of errors caused by incompatible references within the current block. The variable reference must be consistent with the variable declaration.

3.1.3 Combining DECLARE Statements

A separate DECLARE statement is not required for each declaration. For example, instead of writing the two DECLARE statements:

```
    DECLARE CHR BYTE INITIAL ('A');
    DECLARE COUNT INTEGER;
```

Both declarations can be written in a single DECLARE statement, as follows:

```
    DECLARE CHR BYTE INITIAL ('A'), COUNT INTEGER;
```

This declare statement contains two declaration elements, separated by a comma. A declaration element is the text for declaring one identifier (or one factored list of identifiers). Every DECLARE statement contains at least one declaration element. If a DECLARE statement contains more than one declaration element, they are separated by commas.

Most of the examples shown previously have only one declaration element in each DECLARE statement. In the preceding example, the text CHR BYTE INITIAL ('A') is one declaration element; the text COUNT INTEGER is another.

Another way of combining declaration elements is called a factored declaration as indicated above in this section. For example, the non-factored declarations:

```
    DECLARE A BYTE, B BYTE;
    DECLARE C WORD, D WORD;
    DECLARE E DWORD, F DWORD;
```

can be combined as:

```
    DECLARE (A,B) BYTE, (C,D) WORD, (E,F) DWORD;
```

In each factored declaration, the allocated locations are contiguous. Elements declared in a nonfactored declaration statement are not necessarily contiguous.

Variables declared in a factored declaration (i.e., variables within a parenthesized list that are not based, are not used as parameters, or are not EXTERNAL), are stored contiguously in the order specified. (If a based variable occurs in a parenthesized list, the variable is ignored when storage is allocated.)

The declaration elements in a single DECLARE statement are independent of each other, as if they were declared in separate DECLARE statements.

3.2 Initializations

Initialization guarantees that the variables being initialized have a particular value before program execution begins. Every constant should be initialized. Variables can also be initialized. There are no default values for constants or variables. Of course, variables can be initialized by an assignment statement such as the following:

```
PI = 3.1415927;          /* PI must first be declared REAL */
VAR13 = 10;              /* VAR13 must be declared earlier */
```

However, in PL/M, the compiler can set up these values during the compilation rather than using both instruction space and execution time to initialize variables in the program.

There are two kinds of compile-time initializations: INITIAL, used with variables, and DATA, used for constants. (DATA is explained in greater detail later in this section.) In both initializations, the initialization attribute is placed after the type in the declaration. For example:

```
DECLARE FAMILY WORD INITIAL (2);
```

Additionally, when using a linkage attribute (PUBLIC/EXTERNAL), place the linkage attribute after the type declaration and before the initialization attribute.

INITIAL causes initialization to occur during program loading for variables that have storage allocated for them. Such variables can subsequently be changed during execution (just as any other variable). These variables will not be reinitialized on a program restart.

The following rules apply to both INITIAL and DATA:

- INITIAL and DATA cannot be used together in the same declaration.
- INITIAL can occur only in declarations at the outer level of a module. DATA, however, can occur in declarations at any level.
- No initializations are permitted with based variables (discussed in Section 3.5), formal parameters (discussed in Section 8.1.1), or with the EXTERNAL attribute (discussed in Section 7.2).
- Either INITIAL or DATA can follow use of the AT attribute (discussed in Section 3.6). However, if this use of INITIAL or DATA causes multiple initializations, the result cannot be predicted.
- For PL/M-286 and PL/M-386, INITIAL will not initialize pointer (or address) types with absolute values.

PL/M-386

- The initializing value should fit into the space allocated by the data type. The only exception is initialization of HWORD when the offset is derived with a dot operator. For example:

  ```
  DECLARE HH HWORD INITIAL (.B)
  ```

  In this case, the real offset is truncated to give the lower 16 bits. A warning message is issued when an OFFSET value is truncated.

end PL/M-386

The general form of the INITIAL attribute is as follows:

```
INITIAL (value-list)
```

Where:

value-list	is a sequence of values separated by commas.

Values are taken one at a time from the value list and used to initialize the individual scalars being declared. The initialization is performed in the same manner as an assignment. Initial values for members of an array or structure must be specified explicitly. For character string constants, the characters are taken one at a time to initialize an 8-bit scalar, two at a time to initialize a 16-bit scalar, four at a time to initialize a 32-bit scalar, and eight at a time to initialize a 64-bit scalar.

For PL/M-86 and PL/M-286, each value can be a string of up to four characters (e.g., 'A', 'NO'), or an expresion with the restrictions noted in the following list. (Use byte arrays for longer strings because each element can represent one character.)

The expressions used with the INITIAL attribute have the following restrictions:

- For real variables only: An expression can only be a single floating-point constant which can be used to initialize a REAL scalar only. No operator can be used with PL/M-86 and PL/M-286, but a unary + or − operator can be used with PL/M-386.
- For POINTER variables only: A restricted expression can be a location reference formed with the @ operator, which must refer to a variable already declared or to a constant list.
- For all other types (except SELECTOR): A restricted expression can be a constant expression containing no operators except + or −. A constant expression has only whole-number constants as operands (e.g., 2048, 256 + 5), as explained in Chapter 5. The constant expression is evaluated as if it were being assigned to the scalar being initialized, using the rules described in Chapter 5.

- For OFFSET or WORD variables only: A constant expression containing only the + and − operators, and operands that can be whole-number constants and/or "." location references. If the expression contains a "." location reference, only the + operator can precede it. Any combination of + and − operators can follow the "." location reference. For example: 5 + .xyz − 10.

NOTE

For compatibility with programs written in PL/M-80, PL/M-86/286/386 allows an expression containing a location reference formed with the dot operator.

The declaration:

```
DECLARE THRESHOLD BYTE INITIAL (48);
```

declares the BYTE scalar THRESHOLD and initializes the scalar to a value of 48.

The declaration:

```
DECLARE EVEN (5) BYTE INITIAL (2, 4, 6, 8, 10);
```

declares the BYTE array EVEN and initializes its five scalar elements to 2, 4, 6, 8, and 10, respectively.

The declaration:

```
DECLARE COORD STRUCTURE (HIGH$BOUND WORD,
            VALUE (3) BYTE,
            LOW$BOUND BYTE) INITIAL (302, 3, 6, 12, 0);
```

declares the structure COORD and initializes it as follows:

`COORD.HIGH$BOUND` to 302
`COORD.VALUE(0)` to 3
`COORD.VALUE(1)` to 6
`COORD.VALUE(2)` to 12
`COORD.LOW$BOUND` to 0

If a string occurs in the value list, it is taken apart from left to right and the pieces are stored in the scalars being initialized. One character is stored in each BYTE scalar, two characters in each WORD scalar, and four in each DWORD scalar. For example:

```
DECLARE GREETING (5) BYTE AT (@HI) INITIAL ('HELLO');
```

causes GREETING(0) to be initialized with the ASCII code for H, GREETING(1) with the ASCII code for E, and so on.

All the examples shown previously have had value lists that match up one-for-one with the scalars being declared. The value list can have fewer elements than are being declared. Thus:

```
    DECLARE DATUM (100) BYTE INITIAL (3, 5, 7, 8);
```

will work. The first four elements of the array DATUM are initialized with the four elements in the value list, and the remainder of the array is left uninitialized. However, the value list cannot have more elements than are being declared.

3.2.1 The Implicit Dimension Specifier

Often, when initializing an array, you want the array to have the same number of elements as the value list. This can be done conveniently by using the implicit dimension specifier in place of an ordinary dimension specifier (a parenthesized constant). The implicit dimension specifier has the form:

```
    (*)
```

Also use the implicit dimension specifier to define an external or based array whose precise number of elements is either unknown or insignificant. Thus the declaration:

```
    DECLARE FAREWELL(*) BYTE PUBLIC INITIAL ('GOODBYE, NOW');
```

declares a public BYTE array, FAREWELL, with enough elements to contain the string 'GOODBYE, NOW' (namely 12), and initializes the array elements with the characters of the string. To reference this array in another program module, declare it as follows:

```
    DECLARE FAREWELL(*) BYTE EXTERNAL;
```

See Chapter 7 for more information about PUBLIC and EXTERNAL attributes.

Note that the INITIAL and DATA *value-lists* must not be present when the implicit dimension specifier is used with an external array; otherwise, INITIAL and DATA *value-lists* are required. Also, the LENGTH, LAST, and SIZE built-ins cannot be used on an external array that was declared with the implicit dimension specifier.

The following is an example of an implicit dimension in a based declaration, which is described in Section 3.5:

```
    DECLARE X BASED P(*) BYTE;
```

The implicit dimension specifier cannot be used after the parenthesized list of identifiers in a factored declaration (unless it is declared EXTERNAL). Additionally, in PL/M-386, an implicit dimension specifier cannot be used to specify an array that is a member of a structure.

The implicit dimension specifier can be used with any value list; it is not restricted to strings.

3.2.2 Names for Execution Constants: the Use of DATA

A variable is the name of a single data item intended to be used and altered by a program. If the variable is not altered during execution, it is a constant.

For example, the formula for the circumference of a circle (R x 2 x π) or (radius x 2 x π) could be written in PL/M as:

```
C = R * 2.0 * 3.14159;
```

in which C and R would be variables. The declarations for C and R would have to precede the executable statement, and could appear as:

```
DECLARE (C, R) REAL;
```

If pi is used often enough, simplify writing of statements by using PI to declare a symbolic name with that value as follows:

```
DECLARE PI REAL DATA (3.1415927);
```

An array of constants requires a list of values. For example:

```
DECLARE FIBONACCI(9) BYTE DATA (0,1,1,2,3,5,8,13,21);
```

The form and use of the DATA initialization is identical to that of INITIAL except for the following differences:

- DATA causes storage to be allocated in the program's constant data segment. The content and meaning of the name cannot be changed during execution. The name should never appear on the left-hand side of an assignment statement. (This is not the case with INITIAL.)
- DATA initializations can be used in declarations at any block level in the program. (INITIAL can occur only at the module level (i.e., inside the DO-block that is the module itself), outside any sub-blocks that the module may contain.)
- If the keyword DATA is used in a PUBLIC declaration when compiling with the ROM option (see Section 11.2.19), DATA must also be used in the EXTERNAL

declaration of program modules that reference it. However, no *value-list* can be used since the data is defined elsewhere. (INITIAL cannot be combined with EXTERNAL.)

- Use of the AT attribute (as explained in Section 3.6) forces a name to be associated with a specific memory location, which can defeat the purpose of the DATA initialization. (This will not happen with INITIAL unless the variables and locations are explicitly redefined using multiple ATs.)
- If the first declaration has a data initialization, then the variable that is AT that location is also referred to as DATA (i.e., cannot have a value assigned into it).

3.3 Types of Declaration Statements

3.3.1 Compilation Constants (Text Substitution): The Use of LITERALLY

If the program is large enough to have many declarations, declaring a compilation constant will save time at the keyboard, as follows:

```
DECLARE DCL LITERALLY 'DECLARE';
```

Thereafter, during compilation, every time DCL appears alone (not as part of a word), the full string DECLARE will be substituted by the compiler. Subsequent declarations can be written as follows:

```
DCL AREA REAL;
DCL SIZE WORD;
```

A declaration using the reserved word LITERALLY defines a parameterless macro for expansion at compile-time. Declare an identifier to represent a character string, which will then be substituted for each occurrence of the identifier in subsequent text. This expansion will not take place in strings or constants. The form of the declaration is:

DECLARE *identifier* LITERALLY *'string'*;

Where:

identifier	is any valid PL/M identifier.
string	is a sequence of arbitrary characters (limited by the size of the symbol table) from the PL/M set (except an apostrophe).

An apostrophe can be included in a string by writing it as two consecutive apostrophes.

The following example illustrates another use of LITERALLY:

```
DECLARE TRUE LITERALLY '0FFH', FALSE LITERALLY '0';

DECLARE ROUGH BYTE;
DECLARE (X, Y, DELTA, FINAL) REAL;
. . .
ROUGH = TRUE;
DO WHILE ROUGH;
   X = SMOOTH (X, Y, DELTA);
         /* SMOOTH is a procedure declared elsewhere. */
   IF (X-FINAL) < DELTA THEN
         ROUGH = FALSE;

END;
. . .
```

This example of a LITERALLY declaration defines the Boolean values TRUE and FALSE in a manner consistent with the way PL/M handles relational operators (see Chapter 5). Literal substitution for fixed values makes a program more readable.

LITERALLYs can also be used to declare quantities that are fixed for one compilation, but are subject to change from one compilation to the next. Consider the following example:

```
DECLARE BUFFER$SIZE LITERALLY '32';
DECLARE PRINT$BUFFER(BUFFER$SIZE) WORD;
. . .
PRINT$BUFFER(BUFFER$SIZE - 10) = 'G';
. . .
```

A future change to BUFFER$SIZE can be made in one place, at the first declaration, and the compiler will propagate the change throughout the program during compilation. This eliminates the need to search the program for the occurrences of 32 that are BUFFER$SIZE references and not some other reference to 32.

3.3.2 Declarations of Names for Labels

A label marks the location of an instruction. Labels are permitted only on executable statements, not on declarations.

A name can be declared as a label both explicitly and implicitly. Explicit label declarations are used mainly to enable module-to-module references (see Chapter 7). The three explicit label declarations have the following formats:

- `DECLARE PART3 LABEL;`
- `DECLARE START1 LABEL PUBLIC;      /* for intermodule reference */`
- `DECLARE PHASE2 LABEL EXTERNAL;    /* for intermodule reference */`

The rules for explicit label declarations are discussed in detail in Chapter 7.

In implicit label declarations (used more commonly than explicit label declarations), the name is placed at the very beginning of the executable statement to which the name is supposed to point. For example:

```
START2: ALPHA = 127;
```

This statement defines the label START2 as pointing to the location of the PL/M instruction shown. If this block has no explicit declaration of START2, such as the following:

```
DECLARE START2 LABEL;
```

then the compiler takes the definition of START2 as an implicit declaration as well as a definition, as if the declaration had occurred at the start of the last simple DO or procedure statement. (If there is an explicit declaration, then the actual placement of the label remains simply a definition.)

Labels are used to indicate significant instructions or the starting point of instruction sequences. Labels can be useful reference points for understanding the parts of a program, or targets for the transfer of control during execution (as discussed under GOTO and CALL in Chapter 6).

3.3.3 Results of Label Declarations

Valid label declarations result in the following:

- The declared name can be used to point to an executable instruction.
- The use of the declared name as a variable in its block is disallowed.
- If the label is also defined in its block by appearing in an executable statement, the address of that statement will be assigned as the value of the label.

3.3.4 Declaration for Procedures

To declare a procedure, give its name with a statement of the form:

```
name: PROCEDURE
```

followed optionally by parameters, type and/or attributes. The definition of the procedure then follows. The procedure definition is the set of statements declaring items used in the procedure (including any parameters) and the executable statements of the procedure itself. The definition ends with an END statement, optionally including the procedure name.

The complete declaration of a procedure includes all the statements from the PROCEDURE statement through the END statement. This definition/declaration must appear before the procedure name is used in an executable statement, just as variable and constant names must be declared before their use.

The only exceptions arise when the full definition may appear in another separately compiled module where it is declared PUBLIC, or when a procedure has been declared REENTRANT. A PUBLIC procedure can be used (called) only if the calling module meets the following requirements:

1. The procedure has been declared with the EXTERNAL attribute (so the linker or binder will search for it).
2. Each formal parameter the procedure uses has been declared so the compiler can verify correct usage when this module invokes the procedure. End this local declaration with an END statement.

   For example:

```
SUMMER: PROCEDURE (A, B) EXTERNAL;
        DECLARE A WORD, B BYTE;
END SUMMER;
```

See Chapter 7 for details on intermodule references. See Chapter 8 for details on procedure definition and use.

3.4 Data Types

Data types apply not only to variables, but to every value processed by a PL/M program. This includes values returned by procedures as well as values calculated by processing expressions. Data type specifications determine the value an object can have, how this value is stored in memory, and the operations that can be used on the value.

The PL/M compilers recognize five classes of data, each of which has one or more data types.

There are several unsigned binary number types: BYTE (8-bit number), WORD (16-bit number in PL/M-86 and PL/M-286; 32-bit number in PL/M-386), HWORD (PL/M-386 only; same as PL/M-86/286 WORD); and DWORD (32-bit number in PL/M-86/286; 64-bit number in PL/M-386). The OFFSET type (PL/M-386 only) is a 32-bit number that representes the offset portion of a pointer, which has its own type: POINTER. (The POINTER type itself is recognized by PL/M-86 and -286, as well as by PL/M-386.) Note that the compiler controls WORD32 and WORD16 automate mapping 32- and 16-bit types. These controls are discussed in Chapter 11.

There are four signed integer data types: INTEGER (16-bit number in PL/M-86 and PL/M-286; 32-bit number in PL/M-386); CHARINT (PL/M-386 only; an 8-bit number); SHORTINT (PL/M-386 only; same as PL/M-86/286 INTEGER type).

PL/M-86, -286 and -386 all recognize the floating-point data type REAL, for signed 32-bit numbers.

The data types differ from one microprocessor to the other. Throughout this manual, the data types are referenced according to the data type class. Table 3-2 summarizes the data type classes for the 8086, 80286 and 80386 microprocessors. The PL/M-386 data types are defined for WORD32. See Section 3.7 for a discussion on the PL/M-386 compiler's WORD32/WORD16 mapping.

Although the PL/M-386 compiler assumes a 32-bit word, the PL/M-386 compiler accepts PL/M-286 code as input. PL/M-286 code can take advantage of the 32-bit data type provided by the 80386 microprocessor when compiled with the PL/M-386 compiler.

Table 3-2 Data Types

Microprocessor	Data Type and Value	
	Unsigned Binary Number	Description
8086/80286/80386	BYTE	8-bit number ranging from 0 to 255. Occupies one byte of memory.
8086/80286 80386	WORD HWORD	16-bit number ranging from 0 to 65,535. Occupies two contiguous bytes of memory. The least significant 8 bits are stored in the lower address.
80386	WORD	32-bit number ranging from 0 to 4,294,967,295. Occupies two contiguous HWORDs of memory. The least significant 16 bits are stored in the lower address.
8086/80286	DWORD	32-bit number ranging from 0 to 4,294,967,295. Occupies two contiguous WORDs of memory. The least significant 16 bits are stored in the lower address.
80386	DWORD	64-bit number ranging from 0 to (2**64) −1. Occupies two contiguous WORDs of memory. The least significant 32 bits are stored in the lower address.
80386	OFFSET	32-bit number that represents the offset portion of a POINTER. ADDRESS (supported by PL/M-80 and PL/M-86/286) is equivalent to OFFSET.
Microprocessor	**Signed Integers**	**Description**
8086/80286 80386	INTEGER SHORTINT	16-bit number ranging from −32768 to +32767. Occupies two contiguous bytes of memory. The least significant 8 bits are stored in the low address. Internally stored in two's complement notation.

Table 3-2 Data Types (continued)

Microprocessor	Signed Integers	Description
80386	INTEGER	32-bit number ranging from −2,147,483,648 to +2,147,483,647. Occupies four contiguous bytes of memory. The least significant 16 bits are stored in the low address. Internally stored in two's complement notation. WORD32's LONGINT is equivalent to INTEGER (see Section 3.7).
80386	CHARINT	8-bit number ranging from −128 to +127. Occupies one byte of memory. Internally stored in two's complement notation.
Microprocessor	**Real Numbers**	**Description**
8086/80286/80386	REAL	Signed, floating-point number. Occupies four contiguous bytes of memory.
Microprocessor	**Pointers**	**Description**
8086/80286/80386	POINTER	The value is the address of the memory storage location. Consists of a segment selector portion and an offset portion.
Microprocessor	**Selectors**	**Description**
8086/80286/80386	SELECTOR	The value is equivalent to the segment selector portion of a POINTER. Can be used as the base of a based variable.

3.4.1 Unsigned Binary Number Variables: Unsigned Arithmetic

Unsigned arithmetic is used to perform any arithmetic operation on unsigned binary number variables. All of the PL/M operators can be used with these data types. Arithmetic and logical operations on such variables yield a result of one of the unsigned binary number types, depending on the operation and the operands. Relational operations always yield a true or false result of type BYTE.

With unsigned arithmetic, if a large value is subtracted from a smaller one, the result is the two's complement of the absolute difference between the two values. For example, if a BYTE value of 1 (00000001 binary) is subtracted from a BYTE value of 0 (00000000 binary), the result is a BYTE value of 255 (11111111 binary).

Also, the result of a division operation is always truncated (rounded down) to a whole number. For example, if a PL/M-86/286 WORD (or PL/M-386 HWORD) value of 7 (0000000000000111 binary) is divided by a BYTE value of 2 (00000010 binary), the result is a PL/M-86/286 WORD (or PL/M-386 HWORD) value of 3 (0000000000000011 binary).

When declaring a variable that may be used to hold or produce a negative result, it is advisable to make the variable either a signed integer or real. If the variable is supposed to hold or produce a non-integer, it must be declared as REAL. Use of the appropriate data types will reduce the occurrences of incorrect results from arithmetic operations (see Chapter 5).

3.4.2 INTEGER Variables: Signed Arithmetic

The sign bit is 0 if the INTEGER value is positive or zero, and if the value is negative. The magnitude is given in two's complement notation.

3.4.2.1 Signed Arithmetic

For the 8086 and the 80286 microprocessors, arithmetic operations on INTEGER variables use 16-bit signed integer arithmetic to hold signed intermediate or final results. For the 80386 microprocessor, arithmetic operations on signed variables use 32-bit signed arithmetic to hold signed intermediate or final results. Thus, addition and subtraction always produce mathematically correct results if overflow does not occur. (See also the OVERFLOW control in Chapter 11.) Relational operations are signed arithmetic comparisons that yield a true or false result of type BYTE.

However, as with unsigned binary number operands, division produces only an INTEGER result. The result is rounded toward zero (i.e., down if the result is positive, up if the result is negative).

Only the arithmetic and relational operators can be used with signed operands. Logical operators are not allowed except for constant expressions within cast parentheses (see Chapter 5).

3.4.3 REAL Variables: Floating-Point Arithmetic

The value of a REAL variable is a signed floating-point number that occupies four contiguous bytes of memory, which may be viewed as 32 contiguous bits in the single precision format. The bits are divided into fields as follows:

SIGN	EXPONENT	SIGNIFICAND
31	30 23	22 0

The byte with the lowest address contains the least significant 8 bits of the significand, and the byte with the highest address contains the sign bit and the most significant 7 bits of the exponent field.

The sign bit is 0 if the REAL value is positive or zero, and 1 if the REAL value is negative.

The exponent field contains a value offset by 127. In other words, the actual exponent can be obtained from the exponent field value by subtracting 127. This field is all 0s if the REAL value is zero.

The significand contains the binary digits of the fractional part of the REAL value when this part is represented in binary scientific notation. This field is all 0s if the REAL value is zero.

Operations on REAL operands use signed floating-point arithmetic to yield a result of type REAL. The implementation guarantees that the result of each operation is the closest floating-point number to the mathematical real-number result (if overflow or underflow does not occur). The relational operators and the arithmetic operators +, −, *, and / can be used with REAL operands: the MOD operator and the logical operators are not allowed. Arithmetic operations yield a result of type REAL and relational operations yield a true or false result of type BYTE.

The PL/M compiler extends the utility of the REAL data types by holding intermediate results in the numeric coprocessor's temporary-real format (80-bit). This format preserves 64 bits of precision and the full range of representable numbers. The exponent in this format is 15 bits instead of 8 in the single precision format.

The increased exponent range greatly reduces the likelihood of underflow and overflow, and eliminates roundoff as a source of error until the final assignment of the result is performed. Underflow, overflow, and roundoff errors are probable for intermediate computations as well as in the final result. For example, an intermediate underflow result might later be multiplied by a very large factor, providing a final result of acceptable magnitude.

3.4.4 Examples of Binary Scientific Notation

1. Consider the following binary number (which is equivalent to the decimal value 10.25):

   1010.01B

   The dot (.) in this number is a binary point. The same number can be represented as:

   1.01001B * 2**3

   This is binary scientific notation, with the binary point immediately to the right of the most significant digit. The digits 01001 are the fractional part, and 3 is the exponent. This value would be represented in the single precision format as follows:

   - The sign bit would be 0, because the value is positive.
   - The exponent field would contain the binary equivalent of 127 + 3 = 130.
   - The leftmost digits of the fraction field would be 01001, and the remainder of this field would be all 0s.

     The complete 32-bit representation would be:

     ```
     0 10000010 01001000000000000000000
     ```

     and the contents of the four contiguous memory bytes would be as follows:

     ```
     highest address:  01000001
                       00100100
                       00000000
     lowest address:   00000000
     ```

     Note that the most significant digit is not actually represented, because by definition it is a 1 unless the REAL value is zero. If the REAL value is zero, the entire 32-bit representation is all 0s.

2. Consider the fraction 1/16, or 0.0625. In binary, it is:

```
1.0000B * 2**(-4)
```

In single precision format, the actual exponent −4 would be represented as 123 (127 −4), and the fraction field would contain all 0s.

In the single precision format, the largest possible value for a valid exponent field is 254, which corresponds to an actual exponent of 127. Therefore, the largest possible absolute value for a positive or negative REAL value is:

```
1.11111111111111111111111B * 2**127
```

or approximately 3.37 * 10**38.

The lowest permissible exponent field value for a non-zero REAL value is 1, which corresponds to an actual exponent of −126. Therefore, the smallest possible absolute value for a positive or negative REAL value is:

```
1.0B * 2**(-126)
```

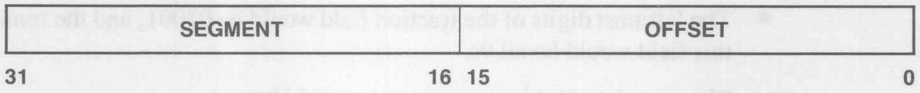
or approximately 8.43 * 10**(−37).

3.4.5 POINTER Variables and Location References

The value of a POINTER variable is the address of the microprocessor's storage location and consists of a segment selector portion (see Chapter 9) and an offset portion.

PL/M-86

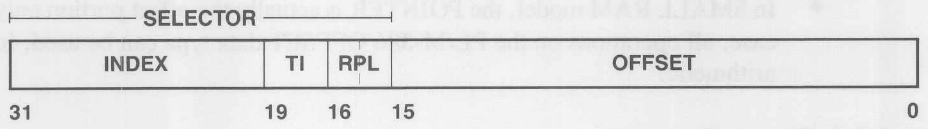
For the 8086 microprocessor, the bits are divided as follows:



end
PL/M-86

PL/M-286

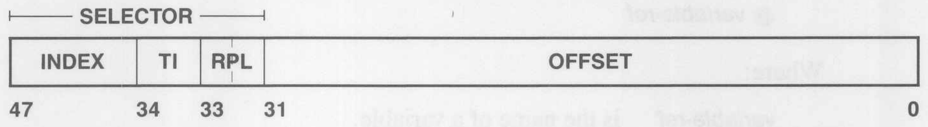
For the 80286 microprocessor, the bits are divided as follows:



end PL/M-286

PL/M-386

For the 80386 microprocessor, the bits are divided as follows:



end PL/M-386

POINTER variables are important as bases for based variables (see Section 3.5).

Only the relational operators for equality and inequality (= or < >) can be used with POINTER operands, yielding a true or false result of type BYTE. No arithmetic or logical operations are allowed (see Chapter 5).

In PL/M-386, a POINTER can be viewed as a structure of SELECTOR and OFFSET rather than a scalar. Therefore, arithmetic with POINTERS (e.g., PTR + 1) is illegal.

The value of a POINTER variable can be created or changed in the following ways:

- The variable can be initialized when declared, using INITIAL or DATA with an address created with @.
- For PL/M-86, the variable can be assigned a whole-number constant (see Chapter 5).
- The variable can be assigned an address created via the @ operator (described in the following section). This is the most commonly used method.
- The variable can be assigned the value of a POINTER variable or function (including NIL, described in Chapter 9).

- The variable can be assigned a value generated by the BUILD$PTR function (also described in Chapter 9).
- POINTER type conversion (cast). Changing from one value to another is different from the POINTER built-in function (see Chapter 9).
- In SMALL RAM model, the POINTER is actually the offset portion only. In this case, all operations on the PL/M-386 OFFSET data type can be used, including arithmetic.

3.4.5.1 The @ Operator

A location reference is formed with the @ operator. A location reference has a value of type POINTER, that is, a location address. An important use of location references is to supply values for POINTER variables.

The basic form of a location reference is as follows:

@ variable-ref

Where:

variable-ref	is the name of a variable.

The value of this location reference is the actual run-time location of the variable.

The *variable-ref* may also refer to an unqualified array or structure name. The pointer value is the location of the first element or member of the array or structure.

Consider the following declarations:

```
DECLARE RESULT REAL;
DECLARE XNUM(100) BYTE;
DECLARE RECORD STRUCTURE (KEY BYTE,
                INFO(25) BYTE,
                HEAD POINTER);
DECLARE LIST(128) STRUCTURE (KEY BYTE,
                INFO(25) BYTE,
                HEAD POINTER);
```

The @RESULT is the location of the REAL scalar RESULT, and @XNUM(5) is the location of the 6th element of the array XNUM. @XNUM is the location of the beginning of the array, that is, the location of the first element (element 0).

The RECORD STRUCTURE declares a byte called KEY followed by 25 bytes called INFO(0), INFO(1), and so on, followed by the POINTER variable named HEAD. Because KEY, INFO, and HEAD are all declared part of the RECORD structure, their contents must be referred to as RECORD.KEY, RECORD.INFO(0), . . . , RECORD.INFO(24), and RECORD.HEAD.

Refer to the addresses of these elements of the RECORD structure by using the @ operator. @RECORD.HEAD is the location of the POINTER scalar RECORD. HEAD and @RECORD is the location of the structure, which is the same as that of the BYTE scalar RECORD.KEY. @RECORD.INFO is the location of the first element of the 25-byte array RECORD.INFO, whereas @RECORD.INFO(7) is the location of the 8th element of the same array.

LIST is an array of structures. The location reference @LIST(5).KEY is the location of the scalar LIST(5).KEY. Note that @LIST.KEY is illegal because it does not identify a unique location (i.e., the KEY of which LIST).

The location reference @LIST(0).INFO(6) is the location of the scalar LIST(0). INFO(6). Also, @LIST(0).INFO is the location of the first element of the same array (i.e., the location of the array itself).

A special case exists when the identifier used as *variable-ref* is the name of a procedure. This procedure must be declared at the outer level of the program module. No actual parameters can be given (even if the procedure declaration includes formal parameters). The value of the location reference in this case is the location of the entry point of the procedure. (Further discussion of procedures is in Chapter 8 and Appendixes F and G.)

3.4.5.2 Storing Strings and Constants via Location References

Another form of location reference is the following:

@(*constant list*)

Where:

constant list	is a sequence of one or more BYTE constants separated by commas and enclosed by parentheses.

When this type of location reference is made, space is allocated for the constants. The constants are stored in this space (contiguously, in the order given by the list), and the value of the location reference is the location of the first constant. If RAM is specified on the compiler invocation command, constants are stored in the DATA segment. If ROM is specified on the compiler invocation command, constants are placed in the CODE segment (see Chapters 11 and 13).

Values in the constant list are treated as if they were in a BYTE array initialization list.

Strings can be included in the list. For example, if the operand:

```
@('NEXT VALUE')
```

appears in an expression, it causes the string 'NEXT VALUE' to be stored in memory (one character per byte, thus occupying 10 contiguous bytes of storage). The value of the operand is the location of the first of these bytes; in other words, it is a pointer to the string.

3.4.6 OFFSET Data Type and the Dot Operator

A dot operator is provided for compatibility with PL/M-80 programs. The dot operator (.) is similar to the @ operator, but produces an address of type WORD. This address represents an offset in the current data segment (for variables) or in the current code segment (for procedures). Use this address with caution, because it can produce unexpected results in a PL/M program that contains more than one data segment or more than one code segment.

In a PL/M-386 program, wherever WORD can be used, OFFSET can also be used. The main difference between the two types is in casting.

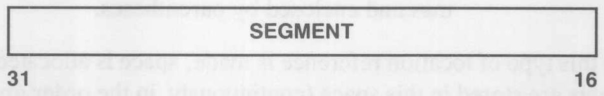
To create or change the value of an OFFSET variable, it can be assigned an OFFSET variable or function, or assigned the result of the built-in function OFFSET$OF, or OFFSET type conversion, or the dot operator (see Chapter 9).

3.4.7 SELECTOR Variables

The value of a SELECTOR variable is equivalent to the segment selector portion of a POINTER, and can also be used as the base of a based variable (see Section 3.5).

PL/M-86

In the PL/M-86, the bits of the SELECTOR portion of a POINTER are as follows:



end
PL/M-86

PL/M-286/386

In PL/M-286 and PL/M-386, the bits of the SELECTOR portion of a POINTER are as follows:

	INDEX	TI	RPL
31	--PL/M-286--	18	16
47	--PL/M-386--	34	32

end PL/M-286/386

The sections of this diagram are discussed in detail in Chapter 10.

Only the logical and relational operators for equality and inequality (=, <, > and < >) can be used with SELECTOR operands, yielding a true or false result of type BYTE. No arithmetic operations are allowed (see Chapter 5).

To create or change the value of a SELECTOR variable it can be assigned a SELECTOR variable or function, or assigned the result of the built-in function SELECTOR$OF or SELECTOR type conversion (see Chapter 9).

The results of the @ and dot operators cannot be assigned directly to SELECTOR variables. They must first be converted to SELECTOR type with the built-in functions SELECTOR$OF and SELECTOR.

3.5 Based Variables

Sometimes, the address of a variable is not known until the program is actually run. For instance, if a procedure is written to swap two bytes and this procedure is called from various places in the code, the addresses of the two bytes are not known when writing the procedure definition.

For this type of manipulation, PL/M uses based variables. A based variable is one that is pointed to by another variable called its base. This means the base contains the address of the desired (based) variable. A variable is made BASED by inserting in its declaration the word BASED and the identifier of the base (which must already have been declared).

A based variable is not allocated storage by the compiler. At different times during program execution the based variable may actually refer to different places in memory, because the variable's base may be changed by the program.

To declare an address based variable, first declare its base, which must be of type POINTER, SELECTOR, or WORD. (Additionally, in PL/M-386, the base can be of type OFFSET.) Next, declare the based variable itself as follows:

```
DECLARE I BYTE;
DECLARE ITEM$PTR POINTER;
DECLARE ITEM BASED ITEM$PTR BYTE;
```

In these declarations, a reference to ITEM is, in effect, a reference to the BYTE value pointed to by the current value of ITEM$PTR. Thus, the sequence:

```
ITEM$PTR = @I;
ITEM = 77H;
```

loads the BYTE value of 77 (hex) into the variable I.

PL/M supports more than one level of based variable, so variables can be based on based variables.

For example, the following declarations are valid:

```
DECLARE PTR1 POINTER;
DECLARE PTR2 BASED PTR1 POINTER;
DECLARE STR1 BASED PTR2 STRUCTURE (
        X REAL,
        Y REAL);
```

The following restrictions apply to bases:

- The base must be of type POINTER, SELECTOR, or WORD (and in PL/M-386, OFFSET). However, use a base of type OFFSET or WORD with caution because it does not contain a full microprocessor address. OFFSET- or WORD-based variables are addressed relative to the current DS register.

- The base cannot be subscripted. That is, it cannot be an array element.

The word BASED must immediately follow the name of the based variable in its declaration, as in the following examples:

```
DECLARE (AGE$PTR, INCOME$PTR, RATING$PTR, CATEGORY$PTR) POINTER;
DECLARE AGE BASED AGE$PTR BYTE;
DECLARE (INCOME BASED INCOME$PTR, RATING BASED RATING$PTR) WORD;
DECLARE (CATEGORY BASED CATEGORY$PTR)(100) WORD;
```

In the first DECLARE statement, the POINTER variables AGE$PTR, INCOME $PTR, RATING$PTR, and CATEGORY$PTR are declared. They are used as bases in the next three DECLARE statements.

In the second DECLARE statement, a BYTE variable called AGE is declared. The declaration implies that whenever AGE is referenced by the running program, its value will be found at the location given by the current value of the POINTER variable AGE$PTR.

The third DECLARE statement declares two based variables, both of type WORD.

The fourth DECLARE statement defines a 100-element WORD array called CATEGORY, based on CATEGORY$PTR. When any element of CATEGORY is referenced at run time, the current value of CATEGORY$PTR is the location of the array CATEGORY (i.e., its first element).

The other elements follow contiguously. The parentheses around the tokens CATEGORY BASED CATEGORY$PTR make the statement more readable, but are not required.

3.5.1 Location References and Based Variables

An important use of location references is to supply values for bases. Thus, the @ operator, together with the based variable concept, gives PL/M a very powerful facility for manipulating pointers.

For example, to refer to the three different REAL variables: NORTH$ERROR, EAST$ERROR, and HEIGHT$ERROR at different times with the single identifier ERROR, write:

```
DECLARE (NORTH$ERROR, EAST$ERROR, HEIGHT$ERROR) REAL;
DECLARE ERROR$PTR POINTER;
DECLARE ERROR BASED ERROR$PTR REAL;
   .  .
   ERROR$PTR = @NORTH$ERROR;
```

The value of ERROR$PTR is the location of NORTH$ERROR. A reference to ERROR is, in effect, a reference to NORTH$ERROR. Later in the program, write:

```
ERROR$PTR = @HEIGHT$ERROR;
```

Now a reference to ERROR is, in effect, a reference to HEIGHT$ERROR. In the same way, the value of the pointer can be made the location of EAST$ERROR, and a reference to ERROR can be made a reference to EAST$ERROR.

This technique is useful for manipulating complicated data structures and for passing locations to procedures as parameters. Examples are given in Chapter 8.

3.6 The AT Attribute

The AT attribute causes the address of a variable to be the specified location. The AT attribute has the form:

AT (*location*)

Where:

location must be a location reference formed with the @ operator (or, in PL/M-86, a whole-number constant in the range 0 to 1,048,575).

AT must refer to a nonbased variable that has already been declared. If there is a subscript expression, it must be a constant expression containing no operators except + and −.

The following are examples of valid AT attributes:

```
AT (@BUFFER)
AT (@BUFFER(128))
AT (@NAMES(INDEX + 1))
```

In the last example, INDEX represents a whole-number constant that has been previously declared with a LITERALLY declaration. The compiler replaces this name with the declared whole-number constant, thus satisfying the restrictions previously mentioned.

PL/M-86/286

NOTE

For compatibility with programs written in PL/M-80, PL/M-86 and PL/M-286 allow the location in an AT attribute to be an expression containing a location reference formed with the dot operator.

end PL/M-86/286

The first nonbased variable in a factored declaration containing the AT attribute will have the address specified by location. Other variables in the same declaration will, in sequence, refer to successive locations thereafter.

For example, the declaration:

```
DECLARE (CHAR$A, CHAR$B, CHAR$C) BYTE AT (@BUFFER);
```

causes the BYTE variable CHAR$A to refer to the location of BUFFER. The variables CHAR$B and CHAR$C are located in the next two bytes after CHAR$A.

The declaration:

```
DECLARE T(10) STRUCTURE (X(3) BYTE,
                                      Y(3) BYTE,
                                      Z(3) BYTE) AT (@DATA$BUFFER);
```

sets up structure references to 90 bytes. They are organized so that each of the 10 members of T refers to nine bytes. The first three use the name X, the second three Y, and the last three Z. Figure 3-1 illustrates this structure.

The preceding declaration, using the AT attribute, causes the beginning of the structure T, namely the scalar T(0).X(0), to be located at the same location as a previously declared variable called DATA$BUFFER. The other scalars making up the structure will follow this location in logical order: T(0).X(1), T(0).X(2), and so on up to T(9).Z(2), which is the last scalar, located in the 89th byte after the location of DATA$BUFFER.

However, no memory locations for these 90 scalars are allocated by this declaration. You determine the contents of the memory space beginning at @DATA$BUFFER.

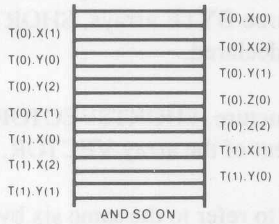


Figure 3-1 Successive Byte References of a Structure

The following rules apply to the AT attribute:

- AT cannot be used with variables that are based, EXTERNAL, or parameters.
- AT can be used with the PUBLIC attribute, if it immediately follows the word PUBLIC. However, the location cannot be a location reference to a variable that is EXTERNAL.

The AT attribute can be used to make variables equivalent, providing more than one way of referring to the same information. For example:

```
DECLARE DATUM WORD;
DECLARE ITEM BYTE AT (@DATUM);
```

causes ITEM to be declared a BYTE variable at the same location that has just been allocated for the WORD variable DATUM. (For the 80386 microprocessor, the preceding example would use HWORD instead of WORD.) Thus, any reference to ITEM is, in effect, a reference to the low-order byte of DATUM (because WORD (or HWORD) values are stored with the low-order 8 bits preceding the high-order 8 bits).

The following is another example using the AT attribute:

```
DECLARE VECTOR (6) BYTE;
DECLARE SHORT$VECTOR STRUCTURE (FIRST (3) BYTE,
                                SECOND (3) BYTE)
                                AT (@VECTOR);
```

In this example, a six-element BYTE array called VECTOR is declared. Additionally, a structure of two three-BYTE arrays, SHORT$VECTOR.FIRST and SHORT-$VECTOR.SECOND is declared.

The first scalar of this structure, SHORT$VECTOR.FIRST(0), is located at the same location as the first element of the array VECTOR.

Thus, there are two ways to refer to the same six bytes. For example, the fifth byte in the group can be referenced as either VECTOR(4) or SHORT$VECTOR.SECOND(1).

When a variable is declared with the AT attribute, the compiler does not optimize the machine code generated to access that variable.

3.7 WORD32/WORD16 Type Mapping

The PL/M-386 compiler supports two primary controls, WORD32 and WORD16, for unsigned binary number and signed integer data types, which provide some basic data type and language semantics compatibility for the 80[x]86 family of microprocessors. These controls specify the basic WORD size and thus affect the representation of certain data types. The default for PL/M-386 is WORD32. This differs from existing 16-bit PL/M-86 and PL/M-286 source code in which the word size is 16 bits. The WORD16 control does not specify 80286 code (e.g., a parameter pushed to the stack is still four bytes), but maps the names of some data types into others. Internally all processing is the same (e.g., signed arithmetic is 32-bit for both WORD16 and WORD32). To accommodate existing 16-bit code where data type representation is critical, WORD16 can be used to map word size to the convention used in PL/M-86 and PL/-286. Table 3-3 lists the data type representation for WORD32 and WORD16.

Table 3-3 WORD32/WORD16 Type Mapping

Unsigned Binary Number Data Types	WORD32 (default)	WORD16
BYTE	8-bit	8-bit
HWORD	16-bit	8-bit
WORD	32-bit	16-bit
DWORD	64-bit	32-bit
QWORD	64-bit	64-bit
Signed Integer Data Types	**WORD32**	**WORD16**
CHARINT	8-bit	8-bit
SHORTINT	16-bit	8-bit
INTEGER	32-bit	16-bit
LONGINT	32-bit	32-bit

NOTE

The PL/M-286 ADDRESS data type is equivalent to WORD. In PL/M-386, ADDRESS is equivalent to the OFFSET data type. OFFSET is a 32-bit data type that represents the offset portion of a POINTER. The size of OFFSET is not affected by the WORD32/WORD16 compiler control.

When writing new PL/M code, or when updating existing PL/M code, it is best to declare variables used for local addressing (i.e., those that are assigned from or initialized to the dot operator (.) location references, assigned from the OFFSET$OF function, or used with the BUILD$PTR function or the STACK$PTR built-in) as OFFSET (or ADDRESS). If the code is compiled using a version of PL/M-86 or PL/M-286 where the OFFSET data type is not part of the language, OFFSET can be declared LITERALLY 'ADDRESS'.

In PL/M-386, WORD is the natural 32-bit data type of the language on which all operations are available. However, in ASM386 a WORD is 16 bits and a DWORD is 32 bits.

3.8 Choosing WORD32 or WORD16

The WORD32/WORD16 compiler control enables determination of how the data type mapping in the source code is interpreted by the PL/M-386 compiler. See Chapter 11 for a description of the WORD32/WORD16 control and syntax.

When compiling new PL/M-386 source code, use WORD32 to take full advantage of the 80386 features.

When recompiling existing PL/M-86 or PL/M-286 code, consider the source code to determine which compiler control to use. WORD32 is usually preferable. Use WORD16 if one of the following conditions apply to the source code:

- Scalar types are mapped to external data, such as STRUCTURES defined to represent data records read from a peripheral device. The format of the data from the peripheral device will not change even though the 80386 microprocessor is processing the data instead of the 8086 or 80286 microprocessors.

- Data is overlaid, for example in a PL/M-286 program:

```
DECLARE W WORD (B1,B2) BYTE AT (@W);
DECLARE P POINTER, B BASED P (2) BYTE, WW BASED P WORD;
```

PL/M-386
(continued)

In the preceding example, code may depend on the fact that two BYTES overlaying the WORD constitute both halves of the WORD completely. Similarly, code can depend on the fact that the LOW or HIGH of a WORD returns 8 bits.

- Loops depend on the size of a WORD type. Operations dependent on a variable overflow could produce unexpected results.

end
PL/M-386

Arrays and Structures 4

Tabs for
452161-001

4 ARRAYS AND STRUCTURES CONTENTS

Page

4 ARRAYS AND STRUCTURES

4.1 Arrays

For increased efficiency, it is often desirable to use a single identifier to refer to a whole group of scalars, and to distinguish the individual scalars by means of a subscript (i.e., a value enclosed in parentheses). Such a list, in which the scalars are all the same type, is called an array.

An array is declared by using a dimension specifier. The dimension specifier is a nonzero whole-number constant enclosed in parentheses. The value of the constant specifies the number of array elements (individual scalar variables) making up the array. For example:

```
DECLARE ITEMS (100) BYTE;
```

causes the identifier ITEMS to be associated with 100 array elements, each of type BYTE. One byte of storage is allocated for each of these scalars.

The elements of an array are stored contiguously, with the first element in the lowest location and the last element in the highest location. No storage is allocated for a based array, but the elements are considered to be contiguous in memory.

The declaration:

```
DECLARE (WIDTH, LENGTH, HEIGHT) (100) REAL;
```

is similar to the following sequence:

```
    DECLARE WIDTH (100) REAL;
    DECLARE LENGTH (100) REAL;
    DECLARE HEIGHT (100) REAL;
```

The difference between the two declarations is that contiguous storage is guaranteed for variables declared in a single parenthesized list, whereas variables declared in consecutive declarations are not necessarily stored contiguously.

This causes each of the three identifiers, WIDTH, LENGTH, and HEIGHT, to be associated with 100 array elements of type REAL, so that 300 elements of type REAL have been declared in all. For each of these scalars, four contiguous bytes of storage are allocated.

4.1.1 Subscripted Variables

To refer to a single element of a previously declared array, use the array name followed by a subscript enclosed in parentheses. This construct is called a subscripted variable.

For example, as a result of the following DECLARE statement:

```
    DECLARE ITEMS (100) BYTE;
```

each byte can be referenced as an individual item using ITEMS(0), ITEMS(1), ITEMS(2), and so on up to ITEMS(99).

Notice that the first element of an array has subscript 0, not 1. Thus, the subscript of the last element is 1 less than the dimension specifier.

To add the third element of the array ITEMS to the fourth, and store the result in the fifth, write the PL/M assignment statement as follows:

```
    ITEMS(4) = ITEMS(2) + ITEMS(3);
```

The subscript of a subscripted variable need not be a whole-number constant. It can be another variable, or any PL/M expression that yields a BYTE, WORD, or INTEGER value for PL/M-86 and PL/M-286, or a BYTE, HWORD, WORD, OFFSET, SHORTINT, CHARINT, or INTEGER value for PL/M-386.

Thus, the construction:

```
    VECTOR(ITEMS(3) + 2)
```

refers to some element of the array VECTOR. Which element this construction refers to depends on the expression ITEMS(3) + 2. This value, in turn, depends on the value stored in ITEMS(3), the fourth element of array ITEMS, at the time when the reference is processed by the running program. If ITEMS(3) contains the value 5, then ITEMS(3) + 2 is equal to 7 and the reference is to VECTOR(7), the eighth element of the array VECTOR.

The following sequence of statements will sum the elements of the 10-element array NUMBERS by using an index variable named I, which takes values from 0 to 9:

```
    DECLARE SUM BYTE;              /* To avoid overflow, */
    DECLARE NUMBERS(10) BYTE;      /* SUM should add up  */
    DECLARE I BYTE;                /* to less than 255   */

    SUM = 0;
    DO I = 0 TO 9;
        SUM = SUM + NUMBERS(I);
    END;
```

Subscripted array variables can be used anywhere a variable can be used, including the left side of an assignment statement if the array elements are of a scalar type.

4.2 Structures

Just as an array enables one identifier to refer to a collection of elements of the same type, a structure enables one identifier to refer to a collection of structure members that may have different data types. Each member of a structure has a member identifier.

A structure member can be another structure; these nested structures are described in Section 4.2.4.

The following is an example of a structure declaration:

```
DECLARE AIRPLANE STRUCTURE (
          SPEED REAL,
          ALTITUDE REAL);
```

This statement declares two REAL scalars, both associated with the identifier AIRPLANE. Once this declaration has been made, the first scalar can be referred to as AIRPLANE.SPEED and the second as AIRPLANE.ALTITUDE. These names are also called the members of this structure.

A structure can have many members (see Appendix B for the correct limit). The members of a structure are stored contiguously in the order in which they are specified. (No storage is allocated for a based structure, but the members are considered to be contiguous in memory.)

Individual structure members cannot be based and cannot have any attributes (see Chapter 3).

4.2.1 Arrays of Structures

With PL/M, arrays of structures can be created. The following DECLARE statement creates an array of structures that can be used to store SPEED and ALTITUDE for 20 AIRPLANEs instead of one:

```
DECLARE AIRPLANE (20) STRUCTURE (
          SPEED REAL,
          ALTITUDE REAL);
```

This statement declares 20 structures associated with the array identifier AIRPLANE, each distinguished by subscripts from 0 to 19. Each of these structures consists of two REAL scalar members. Thus, storage is allocated for 40 REAL scalars.

To refer to the ALTITUDE of the 17th AIRPLANE, write AIRPLANE(16).ALTITUDE.

4.2.2 Arrays within Structures

An array can be used as a member of a structure, as follows:

```
DECLARE PAYCHECK STRUCTURE (
        LAST$NAME(15)BYTE,
        FIRST$NAME(15)BYTE,
        MI BYTE,
        AMOUNT REAL);
```

This structure consists of two 15-element BYTE arrays, PAYCHECK.LAST$NAME and PAYCHECK.FIRST$NAME, the BYTE scalar PAYCHECK.MI, and the REAL scalar PAYCHECK.AMOUNT.

To refer to the fourth element of the array PAYCHECK.LAST$NAME, write PAYCHECK.LAST$NAME(3).

4.2.3 Arrays of Structures with Arrays inside the Structures

Given that an array can be made up of structures, and a structure can have arrays as members, the two constructions can be combined to write:

```
DECLARE FLOOR (30) STRUCTURE (
        OFFICE (55) BYTE);
```

The identifier FLOOR refers to an array of 30 structures, each of which contains one array of 55 BYTE scalars. This could be thought of as a 30-by-55 matrix of BYTE scalars. To reference a particular scalar value (for example, element 46 of structure 25) write FLOOR(24).OFFICE(45). Note that the scalar elements of each OFFICE array are stored contiguously, and the OFFICE arrays are elements of the FLOOR array and are stored contiguously.

Alter the preceding PAYCHECK structure declaration to make it an array of structures, as follows:

```
DECLARE PAYROLL (100) STRUCTURE (
                LAST$NAME(15)BYTE,
                FIRST$NAME(15) BYTE,
                MI BYTE,
                AMOUNT REAL);
```

This is an array of 100 structures, each of which can be used during program execution to store the last name, first name, middle initial, and amount of pay for one employee. LAST$NAME and FIRST$NAME in each structure are 15-byte arrays for storing the names as character strings. To refer to the Kth character of the first name of the Nth employee, write:

```
PAYROLL(N-1).FIRST$NAME(K-1)
```

where N and K are previously declared variables to which appropriate values have been assigned. This might be convenient in a routine for printing out payroll information.

4.2.4 Nested Structures

A member of a structure can also be another structure; this is called a nested structure.

Nested structures are subject to the same rules as all structures. They can contain their own member identifiers, whether these are scalars, arrays, or structures.

The following example shows nested structures:

```
DECLARE EMPLOYEE (100) STRUCTURE (
                ID WORD,
                NAME STRUCTURE (
                   LAST$NAME (15) BYTE,
                   FIRST$NAME (15) BYTE,
                   MI BYTE),
                AGE BYTE,
                JOB WORD,
                PAY STRUCTURE (
                   RATE REAL,
                   OTRATE REAL,
                   BENEFITS STRUCTURE
                      OPTIONS REAL,
                      CHOSEN BYTE)
                          )
                                   );
```

The preceding declaration statement is for an array (named EMPLOYEE) of 100 structures. Each of the 100 elements of EMPLOYEE is a structure with the following members: a WORD scalar named ID, a nested structure called NAME, a BYTE scalar named AGE, a WORD scalar (HWORD for PL/M-386) named JOB, and a nested structure named PAY.

The NAME structure has two arrays (LAST$NAME and FIRST$NAME) of 15 bytes each for members, as well as a BYTE scalar named MI.

The PAY structure has two REAL scalars (RATE and OTRATE) for members, as well as a nested structure named BENEFITS. BENEFITS has the REAL scalar OPTIONS and the BYTE scalar CHOSEN as members.

The preceding example contains two levels of nested structures. The structures NAME and PAY are at the first level of nesting; the structure BENEFITS is at the second level of nesting. See Appendix B for the maximum limit on nested structures.

4.3 References to Arrays and Structures

A variable reference is the use, in program text, of the identifier of a variable that has been declared. A variable reference can be fully qualified, partially qualified, or unqualified.

4.3.1 Fully Qualified Variable References

A fully qualified variable reference specifies a single scalar. For example, given the following declarations:

```
DECLARE AVERAGE REAL;
DECLARE ITEMS (100) BYTE;
   .
   .
   .
DECLARE RECORD STRUCTURE (
        KEY BYTE,
        INFO WORD);
DECLARE NODE (25) STRUCTURE (
        SUBLIST (100) BYTE,
        RANK BYTE);
```

then AVERAGE, ITEMS(5), RECORD.INFO, and NODE(21).SUBLIST(32) are all fully qualified variable references. Each refers unambiguously to a single scalar.

Note that qualification can only be applied to variables that have been appropriately declared. A subscript can only be applied to an identifier that has been declared with a dimension specifier. A member-identifier can be applied only to an identifier declared as a structure identifier. The compiler flags violations of these rules as errors.

4.3.2 Unqualified and Partially Qualified Variable References

Unqualified and partially qualified variable references can be used only in location references (described in Chapter 3) and in the built-in procedures LENGTH, LAST, and SIZE (described in Chapter 9).

An unqualified variable reference is the identifier of a structure or an array, without a member-identifier or subscript. For example, with the declarations in the previous section, ITEMS and RECORD are unqualified variable references. An unqualified variable reference is a reference to the entire array or structure. @ITEMS is the location of the entire array ITEMS (the location of its first byte). Similarly, @RECORD is the location of the first byte of the structure RECORD.

A partially qualified variable reference does not refer to a single scalar even using a subscript and/or member-identifier with an identifier.

For example, in the declaration in the previous section, NODE(15) and NODE(12).SUBLIST are partially qualified variable references.

When used with the @operator, partially qualified variable references are taken to mean the first byte that fits the description. Thus, @NODE(15) is the location of the first byte of the structure NODE(15), which is an element of the array NODE. Similarly, @NODE(12).SUBLIST is the location of the first byte of the array NODE(12).SUBLIST, which is a member of the structure NODE(12), which is an element of the array NODE.

Because it is ambiguous, @NODE.SUBLIST cannot be used. In a location reference referring to an array consisting of structures, a subscript must be given before a member-identifier can be added to the reference. The rule is different for partially qualified variable references in connection with the built-in procedures LENGTH, LAST, and SIZE, as explained in Chapter 9.

Expressions and Assignments

5

Tabs for
452161-001

5 EXPRESSIONS AND ASSIGNMENTS CONTENTS

5 EXPRESSIONS AND ASSIGNMENTS

A PL/M expression consists of scalar operands (values) combined by arithmetic, logical, and relational operators. For example:

```
A + B
A + B − C
A*B + C/D
A*(B + C) − (D − E)/F
```

where +, −, *, and / are arithmetic operators for addition, subtraction, multiplication, and division, and A, B, C, D, E, and F represent operands. The parentheses group operands and operators to control the order of evaluation.

This chapter describes the rules governing PL/M expressions. Although these rules may appear complex, most of the expressions used in actual programs are simple. In particular, when the operands of arithmetic and relational operators are all of the same type, the resulting expression is easy to understand.

5.1 Operands

Operands are the building blocks of expressions. An operand is a quantity with a value at run time on which an arithmetic, logical, or relational operation is performed by an operator. In the preceding examples, A, B, C, etc., are identifiers of scalar variables that have values at run time.

Operands in expressions can also be numeric constants and fully qualified variable references. The following sections describe all of the types of operands that are permitted.

5.2 Constants

A numeric constant can be an operand in an expression. However, its type must be appropriate, as discussed in the following paragraphs.

A numeric constant that contains a decimal point is of type REAL. A numeric constant that does not contain a decimal point is a whole-number constant.

You can use a whole-number constant in either signed context or unsigned context. In unsigned context, a whole-number constant is treated as an unsigned binary number data type. In signed context, a whole-number constant is treated as a signed integer data type. (See Table 3-2.)

5.2.1 Whole-Number Constants in Unsigned Context

PL/M-86/286

In PL/M-86 and PL/M-286, a whole-number constant in unsigned context is treated as follows:

- As a BYTE value if it ranges from 0 to 255
- As a WORD value if it ranges from 256 to 65,535
- As a DWORD value if it ranges from 65,536 to 4,294,967,295 (i.e., 2**32 – 1)

In PL/M-86 only, a single whole-number constant can also be treated as a POINTER or SELECTOR value.

end PL/M-86/286

PL/M-386

In PL/M-386, a whole number constant in unsigned context is treated as follows:

- As a BYTE value if it ranges from 0 to 255
- As a HWORD value if it ranges from 256 to 65,535
- As a WORD value if it ranges from 65,536 to 4,294,967,295 (i.e., 2**32 – 1)
- As a DWORD value if it ranges from 2**32 to 2**64 – 1

end PL/M-386

5.2.2 Whole-Number Constants in Signed Context

In signed context, a whole-number constant is always treated as an INTEGER value. In PL/M-86 and PL/M-286, the range is –32,768 to 32,767.

In PL/M-386, the range is –2,147,483,648 to 2,147,483,647. Additionally, small integer values are extended into 32-bit values with no change to the arithmetic value.

5.2.3 String Constants

A string constant containing not more than four characters can also be used as an operand. If a string constant has only one character, it is treated as a BYTE constant whose value is the 8-bit ASCII code for the character. If a string constant is a two-character string, it is treated as a WORD constant in PL/M-86 and PL/M-286, and as an HWORD constant in PL/M-386. The value of the two-character string is formed by stringing together the ASCII codes for the two characters, with the code for the first character forming the most significant 8 bits of the 16-bit number.

PL/M-86/286

In PL/M-86 and PL/M-286, if a string constant is a three- or four-character string, it is treated as a DWORD constant whose value is formed by stringing together the ASCII codes for all of the characters. In a three-character string, the most significant 16 bits of the 32-bit number are formed of 8 high-order zeros, then the code for the first character. In a four-character string, the code for the first two characters forms the most significant 16 bits of the number.

end PL/M-86/286

PL/M-386

In PL/M-386, if a string constant is a three- or four-character string, it is treated as a WORD constant whose value is formed by stringing together the ASCII codes for all of the characters. The first character represents the high 8 bits, the second character represents the second most significant 8 bits, etc. If the string has three characters, the ASCII NUL character is inserted in front of the first character to form a four-character string.

end PL/M-386

Strings of more than four characters are illegal as operands in expressions, and can be used in only two contexts: as initialization values for an array or as part of a location reference that points to the location at which the string constant is stored (see Chapter 3).

5.3 Variable and Location References

As described in Chapter 4, fully qualified variable references uniquely specify a single scalar value. (Partially qualified references, also discussed in Chapter 4, have very restricted uses.) Any fully qualified variable reference can be used as an operand in an expression. When the expression is evaluated, the reference is replaced by the value of the scalar.

A function reference is the name of a typed procedure that has been declared previously, along with any parameters required by the procedure declaration. The value of a function reference is the value returned by the procedure.

For example, in the statement:

```
    I = J + ABS(L);
```

the absolute value of L will be returned by the function ABS and then added to the value of J before being stored in I. If L is −27, the result will be the same as writing:

```
    I = J + 27;
```

For a complete discussion of procedures and function references, see Chapter 8. Location references are described in Chapter 3.

5.4 Subexpressions

A subexpression is an expression enclosed in parentheses, which can be used as an operand in an expression. A subexpression can be used to group portions of an expression together, just as in ordinary algebraic notation.

5.5 Compound Operands

All the operand types previously described are primary operands. An operand can also be a value calculated by evaluating some portion of the total expression. For example, in the expression:

```
    A + B*C
```

(where A, B, and C are variable references), the operands of the * operator are B and C. The operands of the + operator are A, and the result of the compound operand B * C. Notice that this expression is evaluated as if it had been written as follows:

```
    A + (B * C)
```

This analysis of an expression to determine which operands belong to which operators, and which groups of operators and operands form compound operands, is discussed in Section 5.9. Table 5-1 lists operator precedence.

5.6 Arithmetic Operators

PL/M has the following five principal arithmetic operators:

```
+ - * / MOD
```

These operators are used as in ordinary algebra to combine two operands. Each operand can have an unsigned binary number data type value; a signed integer data type value; or a REAL number data type value (except that REAL operands cannot be used with the MOD operator).

Arithmetic operations cannot be used with POINTER and SELECTOR variables.

Table 5-1 Operator Precedence

Operator Class	Operator	Interpretation
Precedence	( )	Controls order of evaluation: expressions within parentheses are evaluated before the action of any outside operator on the parenthesized items
Unary	+, −	Single positive operator, single negative operator
Arithmetic	*,/,MOD +, −	Multiplication, division, modulo (remainder) division, addition, subtraction
Relational	<, <=, <>, =, >=, >	Less than, less than or equal to, not equal to, equals, greater than or equal to, greater than
Logical	NOT AND OR,XOR	Logical negation Logical conjunction Logical inclusion disjunction, Logical exclusive disjunction

5.6.1 The +, −, *, and / Operators

The operators +, −, *, and / perform addition, subtraction, multiplication, and division on operands of any data type except the POINTER and SELECTOR data types. The following rules govern these operations.

1. Both operands must be of the same class (i.e., both operands must be unsigned, signed, or real). Mixing operands of different classes is illegal. However, an operand of one class can be converted, in an expression, to another class with the use of a built-in conversion function (see Chapter 9).

2. Unsigned Arithmetic:

   Unsigned Addition and Subtraction

   If both operands are of the same data type, the result is of the same data type (e.g., BYTE + or − BYTE produces a BYTE result).

   If the operands are of different data types, the smaller operand is extended with high-order 0 bits to the size of the larger operand; the addition or subtraction is then performed as though both operands are of the same type. For example, for BYTE + or − WORD, the BYTE operand is zero-extended by 8 bits to WORD size; then the operation is performed with the WORD operands to produce a WORD result. For a BYTE + or − DWORD the BYTE operand is zero-extended by 24 bits to DWORD size; then the operation is performed with the DWORD operands to produce a DWORD result. For WORD + or − DWORD, the WORD operand is zero-extended by 16 bits to DWORD size; then the operation is performed with two DWORD operands producing a DWORD result.

   Unsigned Multiplication and Division

PL/M-86/286

   If both operands are of the same type, the result is of the same type as the operands with one exception: if both operands are of type BYTE, the * and / operations produce results of type WORD.

end PL/M-86/286

PL/M-386

Assuming WORD32, if both operands are of type BYTE, the * and / operations produce an HWORD result; if both operands are of type HWORD, the * and / operations produce a WORD result. If both operands are of type WORD, the * and / operations produce a WORD result. If both operands are of type OFFSET, the * and / operations produce an OFFSET result. If both operands are of type DWORD, the * and / operations produce a DWORD result.

For mixed unsigned operands, the same rules as for addition and subtraction apply. The smaller operand is zero-extended to the size of the larger operand, then the multiplication or division is performed as though both operands are of the same type. The results are as described in the preceding paragraph.

If one operand is a whole-number constant or a string constant, it is treated as a HWORD or WORD depending on its value (see Sections 5.2.1 and 5.2.3).

end PL/M-386

PL/M-86/286

3. Signed Arithmetic

All arithmetic for signed operands is signed integer arithmetic. If the whole-number constant exceeds 32,767, the operation is invalid.

During signed arithmetic an expression can overflow only if it overflows the machine word. Assignment overflow is detected using the OVERFLOW control (see Chapter 11).

Constants are always represented as integer constants, regardless of their value.

end PL/M-86/286

PL/M-386

All arithmetic for signed operands is 32-bit signed integer arithmetic. The names of the storage type (e.g., CHARINT) do not imply what type of arithmetic is performed, only the size of storage assigned for the variable.

PL/M-386
(continued)

During signed arithmetic an expression can overflow only if it overflows the machine word (32 bits). However, overflow is possible when the value is assigned to a variable with SHORTINT or CHARINT data type. If the value is assigned to CHARINT, 24 high-order bits are truncated to form the CHARINT value. If the value is assigned to SHORTINT, 16 high-order bits are truncated to form the SHORTINT value. If the value is assigned to INTEGER, it is not changed.

Assignment overflow is detected using the OVERFLOW control (see Chapter 11).

Constants are always represented as integer constants, regardless of their value.

end
PL/M-386

4. Real Arithmetic

   Both operands are always of type REAL. Thus, the +, −, *, and / operations produce a result of type REAL.

   If one operand is a constant, it must be typed as a floating-point constant, that is, it must have a decimal point. Mixing REAL operands with whole-number constants is not allowed. For example, if R is a REAL variable, R + 1.0 is a legal expression, but R + 1 is illegal. Also, 1.0 + 1 is illegal, because it mixes a REAL constant with a whole-number constant.

5. Arithmetic expressions containing operands of type SELECTOR or POINTER are illegal.

6. If both operands are whole-number constants, the operation depends on the context in which it occurs, as explained in Section 5.10.1

7. The result of division by 0 is undefined, except for REAL values (see Appendix G).

A unary − operator is also defined in PL/M. It takes a single operand, to which it is prefixed. A minus sign that has no operand to the left of it is taken to be a unary minus.

A unary − operator makes (− A) equivalent to (0 − A), where A is any operand. The 0 is a BYTE value if A is an unsigned binary number data type. The 0 is an INTEGER value if A is a signed integer data type; or a REAL value if A is a real number data type. If A is a whole-number constant, its type and the unary − operation depend on the context as explained in Section 5.10.1. In unsigned context, (− 1) is assigned a BYTE value (0 – 1) which is equivalent to 0FFH. In signed context, (– 1) is assigned an INTEGER value (0 – 1) which is equivalent to 0FFFFH for PL/M-86 and PL/M-286 and 0FFFFFFFFH for PL/M-386.

Finally, a unary + has no effect; (+ A) is equivalent to (A).

5.6.2 The MOD Operator

MOD performs division, except the result is not the quotient, but rather the remainder left after integer division. The result has the same sign as the operand on the left side of the MOD operator.

REAL operands cannot be used with the MOD operator; only unsigned or signed operands can be used.

For example, if A and B are INTEGER variables with values of 35 and 16, respectively, then A MOD B yields an INTEGER result of 3, and – A MOD B yields – 3.

Unlike the / operator, the MOD operator must be separated from surrounding letters and digits by blanks or other separators.

5.7 Relational Operators

The following relational operators are used to compare operands of the same type:

<	less than
>	greater than
< =	less than or equal to
> =	greater than or equal to
< >	not equal to
=	equal to

Relational operators are always binary operators, taking two operands, to yield a BYTE result. Relational operators can be used with all types.

If both operands are unsigned, then unsigned arithmetic will be used to compare the two values. As with the arithmetic operators, mixing unsigned data types is allowed, with the smaller operand being zero-extended to the size of the larger operand.

Whole-number and string constant operands are treated as BYTE, WORD, or DWORD for PL/M-86 and PL/M-286, and as BYTE, HWORD, WORD, or OFFSET for PL/M-386 (see Section 5.2.1). Unsigned data types are primarily used to represent positive values. Negative numbers are represented by two's complement in the smallest unsigned data type that can hold the value. For example, -2 is represented as the BYTE value of 0FEH. If B is a BYTE variable, then the relational expression B > = −2 is TRUE only if B has the value of 254 or 255, because the expression −2 (when evaluated unsigned) has a BYTE value of 254.

In PL/M-86 and PL/M-286, if both operands are signed, then signed 16-bit integer arithmetic is used to compare the two values. In PL/M-386, if both operands are signed, then signed 32-bit integer arithmetic is used to compare the two values. CHARINT or SHORTINT are sign extended to INTEGER values. The calculated value is then assigned to the specified data type.

If both operands are real, floating point arithmetic will be used to compare the two values. Only floating-point constants (i.e., constants containing a decimal point) can be mixed with REAL operands.

Two POINTER operands can be compared for equality but greater than, less than, and inequality operations cannot be used. In PL/M-386, only two POINTERS are equal only if they are bitwise equal (i.e., if both segment selector portions are equal and both offset portions are equal).

Two SELECTOR operands can be compared for equality, inequality, less than, and greater than.

Since constants cannot be typed as POINTER or SELECTOR, comparison between POINTER or SELECTOR and constants is illegal.

As with arithmetic operations, operands of different classes cannot be mixed together in relational operations. An operand of one class can be converted, in an expression, to another class using a built-in conversion function (see Chapter 9).

If the specified relation between the operands is true, a BYTE value of 0FFH (or 1111$1111B) is returned. Otherwise, the result is a BYTE value of 00H (or 0000$0000B). Thus, in all cases, the result is of type BYTE, with all 8 bits set to 1 for a true condition, or to 0 for a false condition. For example:

```
(6>5)  result is 0FFH ("true")
(6<=4)  result is 00H ("false")
```

Values of true and false resulting from relational operations are useful in conjunction with DO WHILE statements and IF statements, as described in Chapter 6. In the context of a DO WHILE statement or IF statement, only the least significant bit of a TRUE or FALSE value is used. Thus, each value with the least significant bit set (including 0FFH) is considered TRUE and each value with the least significant bit 0 is considered FALSE. A BYTE value is returned.

5.8 Logical Operators

PL/M has the following four logical (Boolean) operators:

```
NOT AND OR XOR
```

These operators are used with the unsigned binary number data type, or whole-number or string constant operands to perform logical operations on 8, 16, or 32 bits.

NOT is a unary operator, taking one operand only. It produces a result of the same type as its operand: each bit of the result is the one's complement of the corresponding bit of the original value.

The remaining operators (AND, OR, XOR) each take two operands, and perform bitwise and, or, and exclusive or, respectively. The bits of an AND result are 1 only when the corresponding bit in each operand is 1. The bits of an OR result are 1 when the corresponding bit of either operand is 1, and 0 only when both operands are 0. The bits of an XOR result are 0 only when the corresponding bits of the operand are the same (i.e., both 1 or both 0); the result has a 1 when one operand is 1 and the corresponding bit of the other operand is 0.

When both operands are of the same type, the result is the same type as the operands.

As with the arithmetic and relational operators, unsigned data types can be mixed in any combination for logical operations. Whole-number operands are treated as BYTE, WORD, or DWORD values in PL/M-86 and PL/M-286, and as BYTE, HWORD, or WORD values in PL/M-386 (see Section 5.2.1). The only exception is an expression composed only of whole numbers within the cast parentheses; then the constants have integer context and the numbers are extended to the 16-bit signed value. The usual bitwise logical operation then takes place (as explained above for 16-bit numbers for bitwise operations). In PL/M-386 only, mixing OFFSET with WORD produces an OFFSET result.

The following are examples of logical operations:

```
NOT 11001100B                result is 00110011B
10101010B AND 11001100B      result is 10001000B
10101010B OR 11001100B       result is 11101110B
10101010B XOR 11001100B      result is 01100110B
```

Note that true and false values resulting from relational operations can be combined with logical operators:

`NOT(6>5)`	result is 00H (false)
`(6>5) AND (1>2)`	result is 00H (false)
`(6>5) OR (1>2)`	result is 0FFH (true)
`(LIM = Y)XOR(Z<2)`	result is 0FFH (true) if LIM = Y and Z⩾2 or if LIM< >Y and Z<2, but result is 00H (false) if both relations are true or both false

Note that in the statement:

```
A = (NOT B)
```

parentheses must be used as indicated. Failure to do so will result in a syntax error because relational operators (=) have higher precedence than logical operators (NOT).

The following are examples of whole numbers. In this example the parentheses enclose items to be converted (casted).

OFFSET(10101010B AND 11001100B) gives OFFSET(010001000B). This is the result obtained with the simple logical operation written above, except that the offset type is returned (in addition, PL/M-386 will extend the answer to 32 bits).

For PL/M-86 and PL/M-286, WORD(−4 AND 7) gives WORD(0FFFCH AND 0007H) which gives WORD(4) or the unsigned 16-bit value of 4.

For PL/M-386, HWORD (−4 AND 7) gives HWORD (0FFFFFFFCH AND 00000007H) which gives HWORD(4) or the unsigned 16-bit value of 4.

5.9 Expression Evaluation

5.9.1 Precedence of Operators: Analyzing an Expression

In PL/M, operators have an implied order that determines how operands and operators are grouped and analyzed during compilation.

The PL/M operators are listed in Table 5-1 from highest to lowest precedence; those that take effect first are listed first. Operators in the same line are of equal precedence and are evaluated as they are encountered in a left to right reading of an expression.

The order of evaluation in an expression is controlled first by parentheses, then by operator precedence, and finally by left to right order.

The compiler first evaluates operands and operators enclosed in paired parentheses as subexpressions, working from the innermost to the outermost pairs of parentheses. The value of the subexpression is then used as an operand in the remainder of the expression.

Parentheses are also used around both subscripts and the parameters of function or procedure references. These are not subexpressions, but they too must be evaluated before the remainder of the expressions or references can be evaluated at a higher level.

When there is more than one operator in an expression, evaluate the results by beginning with the operator with the highest precedence. If the operators are of equal precedence, evaluate them left to right, as follows:

Example	Reason
`(A + B) * C` is not the same as A + B*C	Parentheses form subexpressions
`A + B * C` means the same as A + (B*C)	Operator precedence
`A/B*C` means the same as (A/B)*C	Left to right, equal precedence

The following are examples of precedence ranking:

```
A + B * C            is equivalent to A + (B * C)
A + B - C * D        is equivalent to (A + B) - (C * D)
A + B + C + D        is equivalent to ( (A + B) + C) + D
A / B * C / D        is equivalent to ( (A / B) * C) / D
A>B AND NOT B<C-1    is equivalent to (A>B) AND (NOT(B<(C-1) ) )
```

In the last four examples, the application of the left-to-right rule for operators with the same precedence is shown. In the second, third, and fifth examples, the left-to-right rule for operators of equal precedence makes no difference in the value of the expression. But in the fourth example, the left-to-right rule is critical.

The following example shows the action of the rules of precedence on a longer expression:

```
(-B + SQRT ( B*B - 4.0 * A * C) )/(2.0 * A)
```

Assume A, B, and C are variables of type REAL, and SQRT is a procedure of type REAL which returns the square root of the value passed to it as a parameter. In this case, the parameter is the expression B*B – 4.0*A*C. Floating point constants (4.0, 2.0) are used rather than whole-number constants (4,2) because it is invalid to combine whole-number constants with REAL variables.

The compiler first analyzes the portions of the expressions within the innermost parentheses, then the procedure parameter and the subexpression 2.0 * A. (The subexpression is also called a compound operand because its result is used in evaluating the whole expression.)

In a left-to-right scan, the two operands of the first * operator are both equal to the value of B. The operands of the second * operator are 4.0 and the value of A. The operands of the third * operator are the results of the second evaluation (i.e., the compound operand 4.0*A) and the value of C. The operands of the fourth * operator are 2.0 and the value of A.

The subexpression 2.0* A is now completely analyzed, but the parameter expression still contains a minus (−) operator that has not been analyzed. The operands of this operator are the result of evaluating B*B and the result of evaluating 4.0*A*C. Once the evaluations are done, the parameter expression is analyzed and its value can be calculated.

This value does not become an operand in the overall expression. It is passed to the procedure SQRT, which returns the square root of the parameter. This returned value then becomes an operand in the remainder of the full original expression:

```
(-B + returned value) / (2.0 * A)
```

Now that the innermost subexpressions have been analyzed and evaluated, a division operator whose left operand must be evaluated further remains. This outer subexpression is −B + the returned square root: there are two operators. The first is a unary minus (−) and its operand is the value of B. The second is the binary plus (+) operator, with two operands: the value of −B and the value of SQRT(B*B − 4.0*A*C). −B has the same meaning as 0 − B, which is to be added to the known value of the square root indicated. The final operator is division (/), whose two operands are known: the value of (− B + SQRT(B*B − 4.0*A*C)) and the value of (2.0*A).

Three important points must be emphasized about expression evaluation, as discussed in the next three sections.

5.9.2 Compound Operands Have Types

Compound operands have types as do primary operands. All of the primary operands used in the preceding example were of type REAL, which results in compound operands of type REAL. It is always valid for all the operands in an arithmetic expression to be of the same type, and the result will be that type also. Combining BYTE values can validly create a WORD or HWORD value. Combining a signed integer data type value always creates an INTEGER value.

In an expression containing mixed data types, any combinations can be used as long as the types belong to the same class (i.e., unsigned binary number, signed integer, real, pointer, or selector). Data types (of the same class) can be mixed as operands in expressions, whether they are constants or variables.

Mixing types of different classes in arithmetic, logical, or relational expressions is invalid. For example, if F and G are INTEGER variables and H and K are REAL variables, then the expressions F > K and H + G are invalid.

Due to operator precedence, some combinations can occur validly in the same expression without being directly combined. In the following logical expression:

```
(F > G AND H < K)
```

the subexpression F > G yields a BYTE value, as does the subexpression H < K. Then the BYTE values are ANDed together. This expression is legal despite an apparent mixing of types. G and H could not be the operands for two reasons:

1. The relational operators are of higher precedence than the AND operator.
2. Only unsigned operands are legal with logical operators.

5.9.3 Relational Operators Are Restricted

In the absence of parentheses denoting a subexpression, the result of a relational operation (comparison) cannot become an operand in another relational operation. Thus, the expression:

```
A <=X <=B
```

is invalid in PL/M because the second < = operator would have to use the result of the first < = operator as one of its operands.

In PL/M the valid expression is as follows:

```
A <= X AND X <= B
```

Parentheses also could have created a valid expression; for example:

```
(A <= X)<=B
```

However, in this expression the result does not have the desired meaning: A < = X becomes a byte of value 0 if A is greater than X, 0FFH if A is not greater than X. Thus, if A is 0, X is 1, and B is 2:

```
    (0 < = 1) <= 2
```

evaluates to:

```
    (0FFH)<=2
```

and yields a FALSE value. This is contrary to the original intention.

5.9.4 Order of Evaluation of Operands

Operators and operands are not bound in the same order as the order in which operands are evaluated.

The rules of analysis specify which operands are bound to each operator. The following example is used to show how operands are bound to operators:

```
    A + B*C
```

B and C are the operands of the * operator, and A and the value of B*C are the operands of the + operator. B and C must be evaluated before the * operation can be performed, and the compound operand B*C must be evaluated before the + operation is performed.

However, it is not obvious whether B will be evaluated before C or vice versa. A could be evaluated before either B or C, and its value stored until the + operation is performed.

The rules of PL/M do not specify the order in which subexpressions or operands are evaluated in each statement. This flexibility enables the compiler to optimize the object code it produces, as described in Chapter 11. In most cases, the order of evaluation makes no difference. However, certain embedded assignments (Section 5.11) or function references (Section 8.2) change the value of an operand in the same expression.

5.10 Choice of Arithmetic: Summary of Rules

As described in Chapter 3, PL/M uses three distinct kinds of arithmetic: unsigned, signed, and floating-point. Whenever an arithmetic or relational operation is carried out, PL/M uses one of these types of arithmetic, depending on the types of the operands.

Tables 5-2 and 5-3 are a summary of the rules that determine which type of arithmetic is used in each case. The tables also list the data type of the result for each kind of arithmetic operation. The notes following the table provide additional information. (See Sections 5.7 and 5.8 for rules governing relational and logical operations.)

In PL/M-386, OFFSET operands are always 32-bit unsigned operands.

In expressions, whole-number constants are always converted to the value of the equivalent data type.

Table 5-2 Summary of Expression Rules for PL/M-86 and PL/M-286

Variable Type	Kind of Arithmetic	Operand Type	Arithmetic Operation	Result	Notes
BYTE WORD DWORD	Unsigned	BYTE w/BYTE	+ or −	BYTE	range: 0-255
			* or / or MOD	WORD	range: 0-65535
		BYTE w/WORD becomes WORD w/WORD	any	WORD	A BYTE operand is first extended with 8 high-order zeros to a WORD value.
		BYTE w/DWORD becomes DWORD w/DWORD	any	DWORD	A BYTE operand is first extended with 24 high-order zeros to a DWORD value.
		WORD w/DWORD becomes DWORD w/DWORD	any	DWORD	A WORD operand is first extended with 16 high-order zeros to a DWORD value.
INTEGER	Signed	INTEGER w/INTEGER	any	INTEGER	range: −32768 to +32767
REAL	Floating Point	REAL w/REAL	+ or − or * or /	REAL	--
POINTER		POINTER w/POINTER	=	BYTE	0 or 0FFH
SELECTOR	Unsigned	SELECTOR w/SELECTOR	=, < >, <, or >	BYTE	0 or 0FFH

Note: POINTER values are compared as full microprocessor addresses. SELECTOR values are compared as 16-bit unsigned numbers.

Table 5-3 Summary of Expression Rules for the PL/M-386

Variable Type	Kind of Arithmetic	Operand Type	Operation	Result	Notes
BYTE HWORD WORD DWORD	Unsigned	BYTE w/BYTE	+ or – * / or MOD	BYTE HWORD	range: 0 to 255 0 to 65,535
		HWORD w/HWORD	+ or – * / or MOD	HWORD WORD	range: 0-65,535 0 to 2**32 – 1
		BYTE w/HWORD becomes HWORD w/HWORD	+ or – * / or MOD	HWORD WORD	BYTE is extended with 8 high-order zeros to an HWORD value
		WORD w/WORD	any arithmetic	WORD	range: 0 to 2**32 – 1
		BYTE w/WORD becomes WORD w/WORD	any arithmetic	WORD	BYTE is extended with 24 high-order zeros to a WORD value
		HWORD w/WORD becomes WORD w/WORD	any arithmetic	WORD	HWORD is extended with 16 high-order zeros to a WORD value
		DWORD w/DWORD	any arithmetic	DWORD	range: 0-2**63 – 1
		BYTE w/DWORD becomes DWORD w/DWORD	any arithmetic	DWORD	BYTE is extended with 56 high-order zeros to a DWORD value
		HWORD w/DWORD becomes DWORD w/DWORD	any arithmetic	DWORD	HWORD is extended with 48 high-order zeros to a DWORD value
		WORD w/DWORD becomes DWORD w/DWORD	any arithmetic	DWORD	WORD is extended with 32 high-order zeros to a DWORD value

Table 5-3 Summary of Expression Rules for the PL/M-386 (continued)

Variable Type	Kind of Arithmetic	Operand Type	Operation	Result	Notes
OFFSET	Unsigned	OFFSET w/OFFSET	any arithmetic	OFFSET	range: 0 to 2**32 – 1
		BYTE w/OFFSET becomes OFFSET w/OFFSET	any arithmetic	OFFSET	BYTE is extended with 24 high-order zeros to an OFFSET value
		HWORD w/OFFSET becomes OFFSET w/OFFSET	any arithmetic	OFFSET	HWORD is extended with 16 high-order zeros to an OFFSET value
		WORD w/OFFSET becomes OFFSET w/OFFSET	any arithmetic	OFFSET	range: 0 to 2**32 – 1
		OFFSET w/DWORD becomes DWORD w/DWORD	any arithmetic	DWORD	OFFSET is extended with 32 high-order zeros to a DWORD value
CHARINT SHORTINT INTEGER	Signed	INTEGER w/INTEGER	+ or – * / or MOD	INTEGER	– 2**31 to +2**31 – 1
REAL	Floating Point	REAL w/REAL	+ – * or /	REAL	
POINTER		POINTER w/POINTER	=	BYTE	0 or 0FFH
SELECTOR	Unsigned	SELECTOR w/SELECTOR	=, < >, <, or >	BYTE	0 or 0FFH

Note: CHARINT and SHORTINT are sign extended to INTEGER before expression evaluation.

The combinations of operands shown in Tables 5-2 and 5-3 are the only usable combinations of arithmetic operations and operands. For example, an operand of the signed integer data type cannot be combined with an operand of the unsigned binary number data type. However, explicit conversion can be coded in-line using the PL/M built-ins described in Chapter 9.

5.10.1 Special Case: Constant Expressions

The rules already given explain expressions like:

```
A + 3 * B
```

where there is a single whole-number constant. However, if there is an expression like:

```
3 - 5 + A
```

then the kind of arithmetic that will be used to evaluate 3 − 5 must be considered, because both operands are whole-number constants.

The answer, in this case, depends on the type of operand A. If A is an unsigned binary number, then 3 − 5 is considered to be in unsigned context. Unsigned arithmetic is used to evaluate 3 − 5, giving a BYTE result of 254. Unsigned arithmetic is then used to add this result to A.

For PL/M-86 and PL/M-286, if A is a signed integer, then 3 − 5 is in signed context. Signed arithmetic is used to evaluate 3 − 5, giving an INTEGER result of −2. Signed arithmetic is then used to add this to A.

For PL/M-386, if A is a signed integer, then 3 − 5 is in signed context. Signed 32-bit arithmetic is used to evaluate 3 − 5. Signed 32-bit arithmetic is then used to add this result to A.

If A is of type REAL, POINTER, or SELECTOR, the expression is illegal.

Any compound operand, subexpression, or expression that contains only whole-number constants as primary operands is called a constant expression. Floating-point constants are of type REAL and are treated as the values of REAL variables.

In this expression:

```
3- 5 + 500 + A
```

3 − 5 is a constant expression that forms part of the larger constant expression 3 − 5 + 500.

If the constant expression is not the entire expression, its value is an operand in the expression. The context is created by the other operand of the same operator.

In the preceding example, suppose the operand A has a BYTE value. Then the constant expression 3 − 5 + 500 is in unsigned context. The constants 3 and 5 are treated as BYTE values, and 500 is treated as a WORD or HWORD value. The operation 3 − 5 gives a BYTE result of 254, and this is extended to a WORD or HWORD value of 254 before adding 500. This results in a WORD or HWORD value of 754. It is exactly as if the expression had been written as follows:

```
754 + A
```

For PL/M-86 and PL/M-286, if A had an INTEGER value, the constant 3 − 5 + 500 would be in signed context; signed arithmetic is used for the operation 3 − 5, which results in an INTEGER value of -2. Then 500 is added, and the INTEGER result is 498 which is added to the value of A.

For PL/M-386, if A had a SHORTINT value, the constant $3+5-500$ would be in signed context; signed 32-bit arithmetic is used for the operation $3-5+500$. The result (498) is added to the value of A to form a 32-bit signed temporary result.

In summary, if the context is created by an unsigned binary number data type operand, the constant expression is in unsigned context. If the context is created by a signed integer data type operand, the constant expression is in signed context. Note that if the context is created by a real number, pointer or selector data type operand, the constant expression is illegal.

If the constant expression is the entire expression, then it is one of the following:

- Constant expression as right-hand part of an assignment statement: context is created by the variable to which the expression is being assigned. Rules are given in Section 5.11.
- Constant expression as subscript of an array variable: evaluated as if being assigned to an INTEGER variable (see Section 5.11).
- Constant expression in the IF part of an IF statement: evaluated as if being assigned to a BYTE variable (see Sections 6.3 and 5.11).
- Constant expression in a DO WHILE statement: evaluated as if being assigned to a BYTE variable (see Sections 6.1.3 and 5.11).
- Constant expression as start, step, or limit expression in an iterative DO statement: evaluated as if being assigned to a variable of the same type as the index variable in the same iterative DO statement (see Sections 6.1 and 5.11).
- Constant expression in a DO CASE statement: evaluated as if being assigned to a WORD variable (see Sections 6.1 and 5.11).
- Constant expression as an actual parameter in a CALL statement or function reference: evaluated as if being assigned to the corresponding formal parameter in the procedure declaration (see Sections 8.2 and 5.11).
- Constant expression in a RETURN statement: evaluated as if being assigned to a variable of the same type as the (typed) procedure that contains the RETURN statement (see Section 5.11).
- Constant expression inside an explicit type conversion (cast built-ins); evaluated as if being assigned to an INTEGER variable, shorter values are extended to 16 bits or 32 bits (see Section 5.11). The only exception is that relational operators can be used and are performed bitwise on 16-bit or 32-bit constant values.

5.11 Assignment Statements

Results of computations can be stored as values of scalar variables. At any given moment, a scalar variable has only one value; however, this value can change with program execution. The PL/M assignment statement changes the value of a variable. Its simplest form is:

variable = expression;

where expression is any PL/M expression, as described in the preceding sections. This expression is evaluated, and the resulting value is assigned to (that is, stored in) the variable. This variable can be any fully qualified variable reference except a function reference. The old value of the variable is lost.

For example, following execution of the statement:

```
RESULT = A + B;
```

the variable RESULT will have a new value, calculated by evaluating the expression A + B.

5.11.1 Implicit Type Conversions

In an assignment statement, if the type of the value of the right-hand expression is not the same as the type of the variable on the left side of the equal sign, then either the assignment is illegal or an implicit type conversion occurs. For PL/M-86 and PL/M-286, all BYTE, WORD, or DWORD values are converted automatically. For PL/M-386, all unsigned binary number, signed integer and real data type values are converted automatically. Chapter 9 includes a description of built-in functions that, when invoked, perform explicit conversions for use in expressions or assignments.

For implicit type conversions, the data type of the value on the right-hand side of the assignment statement is always coerced to equal the data type of the value on the left-hand side of the assignment statement. This is done either by extending the value of the expression, or by truncating the value of the expression by the appropriate number of high-order bits so that the data types of both sides of the assignment statement are the same.

The implicit type conversions that occur for assignment statements are summarized in Tables 5-4 and 5-5.

Table 5-4 Implicit Type Conversions in Assignment Statements for PL/M-86 and PL/M-286

Expression Result Type	Variable on Left of Assignment Statement	Conversion
BYTE	WORD	BYTE value is extended by 8 high-order 0 bits to WORD value
	DWORD	BYTE value is extended by 24 high-order 0 bits to DWORD value
WORD	BYTE	8 high-order bits of WORD value are truncated to convert it to a BYTE value
	DWORD	WORD value is extended by 16 high-order 0 bits to DWORD value
DWORD	BYTE	24 high-order bits of DWORD value are truncated to convert it to a BYTE value
	WORD	16 high-order bits of DWORD value are truncated to convert it to a WORD value

Table 5-5 Implicit Type Conversions in Assignment Statements for PL/M-386*

Expression Result Type	Variable on Left of Assignment Statement	Conversion
BYTE	HWORD	BYTE value is extended by 8 high-order 0 bits to HWORD value
	WORD	BYTE value is extended by 24 high-order 0 bits to WORD value
	DWORD	BYTE value is extended by 56 high-order 0 bits to DWORD value
	OFFSET	BYTE value is extended by 24 high-order 0 bits to OFFSET value
HWORD	BYTE	8 high-order bits of HWORD value are truncated to convert it to a BYTE value
	WORD	HWORD value is extended by 16 high-order 0 bits to convert it to a WORD value

*Assuming WORD32

Table 5-5 Implicit Type Conversions in Assignment Statements for PL/M-386 (continued)*

Expression Result Type	Variable on Left of Assignment Statement	Conversion
HWORD (continued)	DWORD	HWORD value is extended by 48 high-order 0 bits to convert it to a DWORD value
	OFFSET	HWORD value is extended by 16 high-order 0 bits to convert it to an OFFSET value
WORD	BYTE	24 high-order bits of WORD value are truncated to convert it to a BYTE value
	HWORD	16 high-order bits of WORD value are truncated to convert it to a HWORD value
	DWORD	WORD value is extended by 32 high-order 0 bits to convert it to a DWORD value
	OFFSET	No conversion (both WORD and OFFSET are 32-bits)
DWORD	BYTE	56 high-order bits of DWORD value are truncated to convert it to BYTE value
	HWORD	48 high-order bits of DWORD value are truncated to convert it to HWORD value
	WORD	32 high-order bits of DWORD value are truncated to convert it to WORD value
	OFFSET	32 high-order bits of DWORD value are truncated to convert it to OFFSET value
OFFSET**	BYTE	24 high-order bits of OFFSET value are truncated to convert it to a BYTE value
	HWORD	16 high-order bits of OFFSET value are truncated to convert it to a HWORD value
	WORD	No conversion is necessary (both WORD and OFFSET are 32 bits)
	DWORD	OFFSET value is extended by 32-high-order 0 bits to convert it to a DWORD value
INTEGER	CHARINT	24 high-order bits of INTEGER value are truncated to convert it to CHARINT value
	SHORTINT	16 high-order bits of INTEGER value are truncated to convert it to SHORTINT value
REAL	REAL	Automatically converted to 32-bit value

*Assuming WORD32
**A warning message is issued if OFFSET values are truncated.

PL/M-86/286

Expression with an INTEGER value: No implicit conversions are performed. If the variable on the left side of the assignment statement is a type other than INTEGER, the assignment is illegal.

Expression with a REAL value: No implicit conversions are performed. If the variable on the left side of the assignment statement is a type other than REAL, the assignment is illegal.

Expression with a POINTER value: No implicit conversions are performed. If the variable on the left side of the assignment statement is a type other than POINTER, the assignment is illegal.

Expression with a SELECTOR value: No implicit conversions are performed. If the variable on the left side of the assignment statement is a type other than SELECTOR, the assignment is illegal.

end PL/M-86/286

PL/M-386

Note that implicit conversion is not performed for POINTER or SELECTOR values. For assignment statements with POINTER or SELECTOR expressions, the left side of the assignment statement would be of the same type as the expression.

end PL/M-386

5.11.2 Constant Expression

BYTE variable on the left: The constant expression is evaluated in unsigned context. If the resulting value is equal to or greater than 0 and equal to or less than 255, it is treated as a BYTE value and no conversion is necessary. If the resulting value is greater than 255, it is truncated to type BYTE by dropping all except its 8 low-order bits.

INTEGER variable on the left: The constant expression is evaluated in signed context. No conversion is necessary.

REAL variable on the left: The assignment is illegal unless all values on the right are floating-point constants. If the value of the constant expression is out of the range for REAL variables, an overflow exception occurs (see Section 10.6 and Appendix G).

PL/M-86

POINTER variable on the left: If the constant expression consists only of a single whole-number constant, the constant is treated as a POINTER value. The whole-number constant must not be greater than 1,048,575. If the constant expression is a value other than a single whole-number constant, the assignment is illegal. This is one of the three cases in which a whole-number constant can be treated as a POINTER value. The other two cases are described in Sections 3.4.5 and 3.5.

SELECTOR variable on the left: If the constant expression consists only of a single whole-number constant, it is treated as a SELECTOR value. The whole-number constant must not exceed 65,535. If the constant expression is a value other than a single whole-number, the assignment is illegal. This is one of two cases where a whole-number constant can be treated as a SELECTOR value. The other case is described in section 3.4.7.

end PL/M-86

PL/M-86/286

WORD variable on the left: The constant expression is evaluated in unsigned context. If the resulting value is equal to or greater than 0 and equal to or less than 65,535, it is treated as a WORD value, and no conversion is necessary. If the resulting value is greater than 65,535, it is truncated to type WORD by dropping all except its 16 low-order bits.

DWORD variable on the left: The constant expression is evaluated in unsigned context. No conversion is necessary.

end PL/M-86/286

PL/M-286

POINTER variable on the left: The assignment is illegal.

SELECTOR variable on the left: The assignment is illegal.

end PL/M-296

PL/M-386

HWORD variable on the left: The constant expression is evaluated in unsigned context. If the resulting value is equal to or greater than 0 and equal to or less than 65,535, it is treated as an HWORD value, and no conversion is necessary. If the resulting value is greater than 65,535, it is truncated to type HWORD by dropping all except its 16 low-order bits.

WORD variable on the left: The constant expression is evaluated in unsigned context. No conversion is necessary.

DWORD variable on the left: The constant expression is evaluated in unsigned context and is zero-extended to a DWORD value.

OFFSET variable on the left: The constant expression is evaluated in unsigned context. No conversion is necessary.

CHARINT variable on the left: The constant expression is evaluated in 32-bit INTEGER arithmetic. If the value is less than −128 or greater than +127, it is truncated to 8 bits.

SHORTINT variable on the left: The constant expression is evaluated in 32-bit INTEGER arithmetic. If the value is outside the given range for SHORTINT (−32,768 to +32,767), it is truncated to 16 bits.

Constants cannot be assigned to POINTER or SELECTOR variables.

end
PL/M-386

Type conversion built-ins can be used to change the type of a constant expression to the type required for assignment. The entire expression within the type conversion is evaluated in signed context.

5.11.3 Multiple Assignment

It is often convenient to assign the same value to several variables at the same time. This is accomplished in PL/M by listing all the variables to the left of the equal sign, separated by commas. The variables LEFT, CENTER, and RIGHT can all be set to the value of the expression INIT + CORR with the single assignment statement:

```
LEFT, CENTER, RIGHT = INIT + CORR;
```

The variables on the left-hand side of a multiple assignment must be all of the same class, that is, all unsigned, all signed, all pointer, all selector, or all real. Then the conversion rules described previously in this chapter are applied separately to each assignment.

NOTE

The order in which the assignments are performed is not guaranteed. Therefore, if a variable on the left side of a multiple assignment also appears in the expression on the right side, the results are undefined.

5.11.4 Embedded Assignments

A special form of assignment can be used within PL/M expressions. The form of this embedded assignment is:

variable: = *expression*

and can appear anywhere an expression is allowed. The expression (everything to the right of the : = assignment symbol) is evaluated and stored in the variable on the left. Parentheses are used to specify the limits of an embedded assignment within an assignment statement. The value of the embedded assignment is the same as that of its right half. For example, the expression:

```
ALT + (CORR := TCORR + PCORR) - (ELEV := HT/SCALE)
```

results in exactly the same value as:

```
ALT + (TCORR + PCORR) - (HT/SCALE)
```

except that the intermediate results TCORR + PCORR and HT/SCALE are stored in CORR and ELEV, respectively. These names for intermediate results can then be used at a later point in the program without recalculating their values. The names must have been declared earlier.

NOTE

The rules of PL/M do not specify the order in which subexpressions or operands are evaluated. When an embedded assignment changes the value of a variable that also appears elsewhere in the same expression, the results cannot be guaranteed.

For example, the following expression:

```
A = (X:=X+4) + Y*Y + X;
```

could mean either of the following interpretations:

```
A1 = (X+4) + Y*Y + (X+4);
A2 = (X+4) + Y*Y + X;
```

Avoid this ambiguity by removing the embedded assignment from the expression and using a separate assignment statement to achieve the desired effect as follows:

```
1     X = X + 4;
     A1 = X + Y*Y + X;

2     X = X + 4;
     A2 = X + Y*Y + X - 4;

3    A3 = X + 4 + Y*Y + X;
      X = X + 4;
```

Flow Control Statements 6

Tabs for
452161-001

6 FLOW CONTROL STATEMENTS CONTENTS

Page

6 FLOW CONTROL STATEMENTS

This chapter describes statements that alter the sequence of PL/M statement execution and that group statements into blocks.

6.1 DO and END Statements: DO Blocks

Procedures and DO blocks are the basic building units of modular programming in PL/M. (Procedures are discussed in Chapter 8.)

This chapter discusses all four kinds of DO-blocks. Each DO block begins with a DO statement and includes all subsequent statements through the closing END statement. The four kinds of DO blocks are as follows:

- Simple DO block

```
DO;                          /* all statements executed, each in order */
  statement-0;
  statement-1;
  statement-2;
  .
  .
END;
```

- DO CASE block

```
DO CASE select_expression;      /* exactly one statement executed */
  case-0-statement;           /* executed if select_expression = 0 */
  case-1-statement;           /* executed if select_expression = 1 */
  .                                                      /* etc. */
  .
END;
```

- DO WHILE block

```
DO WHILE expression_true;
  statement-0;          /* all executed repeatedly if expression is */
  statement-1;          /* true, none executed if expression false. */
  .
  .
END;
```

- Iterative DO block

```
DO counter = start-expr TO limit-expr BY step-expr;
  statement-0;              /* all statements executed a number */
  statement-1;              /* of times depending on comparison */
                                 /* of counter with limit expr. */
  .
  .
  .
END;
```

The last two blocks are also referred to as DO-loops because the executable statements within them can be executed repeatedly (in sequence) depending on the expressions in the DO statement.

Any DO statement can have multiple labels on it, and only the last of these can appear between the word END and the next semicolon. For example:

```
    A: B: C: D: EM: DO;
                    .
                    .
                    .
                    END EM ;/* indicates end of block EM; */
                                 /* A, B, C, D also end here. */
```

As mentioned in Chapter 3, the placement of declarations is restricted. Except for use in procedures, declarations are permitted only at the top of a simple DO block, before any executable statements of the block. (This DO can, of course, be nested within other DOs or procedures. Chapter 7 discusses the scope of declared names.)

Each DO block can contain any sequence of executable statements, including other DO blocks. Each block is considered by the compiler as a unit, as if it were a single executable statement. This fact is particularly useful in the DO CASE block and the IF statement, both discussed in this chapter.

The discussions that follow describe the normal flow of control within each kind of DO block. The normal exit from the block passes through the END statement to the statement immediately following. These discussions assume that none of the statements in the block causes control to bypass that process. A GOTO statement with the target outside the block would be one such bypass. (GOTOs are discussed later in this chapter.)

6.1.1 Simple DO Blocks

A simple DO block merely groups, as a unit, a set of statements that will be executed sequentially (except for the effect of GOTOs or CALLs):

```
DO;
   statement-0;
   statement-1;
   ...
   statement-n;
END;
```

For example:

```
DO;
    NEW$VALUE = OLD$VALUE + TEMP;
    COUNT = COUNT + 1;
END;
```

This simple DO block adds the value of TEMP to the value of OLD$VALUE and stores it in NEW$VALUE. It then increments the value of COUNT by one.

DO blocks can be nested within each other as shown in the following example:

```
ABLE: DO;
                 statement-0;
                 statement-1;
BAKER:           DO;
                      statement-a;
                      statement-b;
                      statement-c;
                 END BAKER;
                 statement-2;
                 statement-3;
      END ABLE;
```

The first DO statement and the second END statement bracket one simple DO block. The second DO statement and the first END statement bracket a different DO block inside the first one. Notice how indentation (using tabs or spaces) can be used to make the sequence more readable, so that it can be seen at a glance that one DO block is nested inside another. It is recommended that this practice be followed in writing PL/M programs. See Appendix B for the number of DO blocks that can be nested.

A simple DO block can delimit the scope of variables, as discussed in Chapter 7.

6.1.2 DO CASE Blocks

A DO CASE block begins with a DO CASE statement, and selectively executes one of the statements in the block. The statement is selected by the value of an expression. The maximum number of cases is given in Appendix B. The form of the DO CASE block is:

```
DO CASE select_expression;
          statement-0;
          statement-1;
          . . .
          statement-n;
END;
```

In the DO CASE statement, *select_expression* must yield an unsigned binary number (excluding DWORD) or a signed integer value. If the expression is a constant expression, it is evaluated as if it were being assigned to a WORD variable. The value (call the value K) must be between 0 and *n*, inclusive. K is used to select one of the statements in the DO CASE block, which is then executed. The first case (*statement-0*) corresponds to K = 0; the second (*statement-1*) corresponds to K = 1, and so forth. Only one statement from the block is selected. This statement is then executed (only once). Control then passes to the statement following the END statement of the DO CASE block.

NOTE

If the run-time value of the expression in the DO CASE statement is less than 0 or greater than n (where n + 1 is the number of statements in the DO CASE block), the effect of the DO CASE statement is undefined. This may have disastrous effects on program execution. Therefore, if there is any possibility that this out-of-range condition may occur, the DO CASE block should be contained within an IF statement that tests the expression to make sure that it has a value that will produce meaningful results.

An example of a DO CASE block is:

```
DO CASE SCORE;
    ;                                                /* case 0 */
    CONVERSIONS=CONVERSIONS + 1;                     /* case 1 */
    SAFETIES = SAFETIES + 1;                         /* case 2 */
    FIELDGOALS = FIELDGOALS + 1;                     /* case 3 */
    ;                                                /* case 4 */
    ;                                                /* case 5 */
    TOUCHDOWNS=TOUCHDOWNS + 1;                       /* case 6 */
END;
```

When execution of this CASE statement begins, the variable SCORE must be in the range 0 to 6. If SCORE is 0, 4, or 5 then a null statement (consisting of only a semicolon, and having no effect) is executed; otherwise the appropriate statement is executed, causing the corresponding variable to be incremented.

A more complex DO CASE block is the following:

```
SELECT = COUNT - 5;
IF SELECT <= 2 AND SELECT >= 0 THEN
    DO CASE SELECT;

        X = X + 1;                             /* Case 0 */

        DO;                              /* Begin Case 1 */
            X = Y + 10;
            Y = Y + 1;
        END;                               /* End Case 1 */

        DO I = LAST$HI + 1 TO TOP - 6;/* Begin Case 2 */
            Z(I) = X * Y + 1;
            W(I) = Z(I) * Z(I);
            V(I) = W(I) - Z(I);
        END;                               /* End Case 2 */

    END;                           /* End DO CASE block */
ELSE CALL ERROR;
```

If SELECT and COUNT are INTEGER variables, negative values could occur. The DO CASE block is placed within an IF statement to guarantee that execution of the DO CASE block will not be attempted if the value of SELECT is less than 0 or greater than 2. Instead, a procedure called ERROR (declared previously) will be activated. IF statements are discussed in Section 6.3.

The preceding example illustrates the use of a simple DO block as a single PL/M statement. The DO CASE statement can select Case 1 or Case 2 and cause multiple statements to be executed. This is only possible because they are grouped as a simple DO block, which acts as a single statement.

6.1.3 DO WHILE Blocks

DO WHILE and IF statements examine only the least significant bit of the value of the expression. If the value is an odd number (least significant bit = 1), it will be considered true. If it is even (least significant bit = 0), it will be considered false. If the expression is relational, e.g., A < B, the result will have a value of 00H or 0FFH, but this is incidental; it may have any unsigned value.

A DO WHILE block begins with a DO WHILE statement, and has the following form:

```
DO WHILE expression;   /* expression must yield an unsigned value */
         statement-0;
         statement-1;
         . . .
         statement-n;
END;
```

The effect of this statement is as follows:

1. First the unsigned expression following the reserved word WHILE is evaluated. If the rightmost bit of result is 1, then the sequence of statements up to the END is executed.
2. When the END is reached, the expression is evaluated again, and again the sequence of statements is executed only if the value of the expression has a rightmost bit of 1.
3. The block is executed over and over until the expression has a value whose rightmost bit is 0. Execution then skips the statements in the block and passes to the statements following the END statement.

Consider the following example:

```
AMOUNT = 1;
DO WHILE AMOUNT <= 3;
      AMOUNT = AMOUNT + 1;
END;
```

The statement AMOUNT = AMOUNT + 1 is executed exactly 3 times. The value of AMOUNT when program control passes out of the block is 4.

6.1.4 Iterative DO Blocks

An iterative DO block begins with an iteration statement and executes each statement in the block, in order, repeating the entire sequence. The form of the iterative DO block is:

```
DO counter = start-expr TO limit-expr BY step-expr ;
   statement-0 ;
   statement-1 ;
   .
   .
END;
```

The BY *step-expr* phrase is optional; if omitted, a step of 1 is the default.

For PL/M-86 and PL/M-286, the counter must be a non-subscripted variable of unsigned type: BYTE or WORD, or a signed integer data type: INTEGER. For PL/M-386, the counter must be a non-subscripted variable of unsigned type: BYTE, HWORD, WORD, or OFFSET, or a signed integer data type: INTEGER, CHARINT, or SHORTINT.

An example of an iterative DO block is:

```
DO I = 1 TO 10;
      CALL BELL;
END;
```

where BELL is the name of a procedure that causes a bell to ring. The bell will ring ten times.

Another example shows how the index-variable can be used within the block:

```
AMOUNT = 0;
DO I = 1 TO 10;
      AMOUNT = AMOUNT + I;
END;
```

The assignment statement is executed 10 times, each time with a new value for I. The result is to sum the numbers from 1 to 10 (inclusive) and leave the sum (namely, 55) as the value of AMOUNT.

The next example uses *step-expr*:

```
    /* Compute the product of the first N odd integers */
PROD = 1;
DO I = 1 TO (2*N-1) BY 2;
      PROD = PROD*I;
END;
```

The type of counter (signed or unsigned) affects the following factors in the execution flow of iterative DOs:

- When *step-expr* is evaluated.
- What causes execution to exit the DO block.

The following steps constitute the general execution sequence of an iterative DO block, with both signed and unsigned variables and expressions in the DO itself. Type is mentioned only for steps in which actions or consequences vary according to type. Where the signed case is different, it is described in parentheses. The discussion following this description summarizes the rules and their results for signed and unsigned data types.

1. The *start-expr* is evaluated and assigned to counter.
2. The *limit-expr* is evaluated and compared with counter. (If counter and *limit-expr* are of signed type, then *step-expr* is also newly evaluated at this time.)
   a. If counter is greater than *limit-expr*, execution exits the DO and passes to the statement following the next END (unless *step-expr* is a negative signed value; if so, the exit occurs only if counter is less than *limit-expr*).
   b. Otherwise, the statements within the DO block are executed in order until the END statement is reached.
   c. At the END, a *step-expr* of unsigned type (BYTE or WORD for PL/M-86 and PL/M-286, and BYTE, HWORD, or WORD for PL/M-386) is newly evaluated.
3. The counter is incremented by the value of *step-expr*. For unsigned counters, if the new value is less than the old value (due to modulo arithmetic as explained next), the loop is exited immediately. Otherwise, control returns to step 2.

An 8-bit BYTE can represent numbers no larger than 1111111B (255 decimal). The largest number a 16-bit WORD (or HWORD) can represent is 1111111111111111B, which is 65535 decimal. The largest number a 32-bit WORD can represent is 0FFFF$FFFFH, which is 4,294,967,295 decimal. Adding 1 to these values gives a result of 0. Thus, the new counter can be less than the old.

These rules and their consequences can be summarized in two broad cases:

1. Starting with a non-negative *step-expr*, then the loop is exited as soon as any one of the following conditions become true:

   a. The new counter is greater than the new *limit-expr*.

   b. A signed *step-expr* becomes negative AND the new counter is still less than the new *limit-expr*.

   c. An unsigned *step-expr* causes a lower counter than the one just used.

2. When starting with a negative and signed *step-expr*, then the loop is exited as soon as either of the following two conditions occurs:

   a. The new counter is less than the new *limit-expr*.

   b. The new *step-expr* becomes non-negative AND the new counter is greater than the new *limit-expr*.

Upon exit from the iterative DO block:

1. In all cases *step-expr* has been reevaluated.

2. In all but one case *limit-expr* has been reevaluated. When an unsigned counter has just gone over and become smaller, *limit-expr* is unchanged from its value during the last loop.

3. In all cases counter has been changed, but the step value that was added to it varies. If signed, counter has been incremented by the former step value before it was reevaluated. For unsigned counters, the newer step has been used.

The following distinctions are important:

- In every case, *start-expr* is evaluated only once and *limit-expr* is evaluated before any execution.

- A signed *step-expr* is evaluated in step 2; other *step-expr*s are evaluated in step 3.

- With an unsigned counter, there cannot be a negative step. Furthermore, stepping down to a *limit-expr* that is less than *start-expr* is not possible because the loop will be exited immediately.

6.2 END Statement

An END statement must terminate all DO blocks. An END statement has the following syntax:

```
END [name];
```

Where:

name	is the optional name that (if present) should match the label of the corresponding DO statement.

6.3 IF Statement

The IF statement provides conditional execution of statements. It takes the form:

```
IF expression THEN statement-a;
    ELSE statement-b;                   /*optional*/
```

The reserved word THEN and the statement following it are required and are called the THEN part. The reserved word ELSE and the statement following it are optional, and are called the ELSE part.

The IF statement has the following effect: first *expression* is evaluated as if it were being assigned to a variable of type BYTE. If the result is true (rightmost bit is 1) then *statement-a* is executed. If the result is false (rightmost bit is 0), then *statement-b* is executed. Following execution of the chosen alternative, control passes to the next statement following the IF statement. Thus, of the two statements (*statement-a* and *statement-b*) only one is executed.

Consider the following program fragment:

```
IF NEW > OLD THEN RESULT = NEW;
      ELSE RESULT = OLD;
```

Here, RESULT is assigned the value of NEW or the value of OLD, whichever is greater. This code causes exactly one of the two assignment statements to be executed. RESULT always gets assigned some value, but only one assignment to RESULT is executed.

In the event that *statement-b* is not needed, the ELSE part may be omitted entirely. Such an IF statement takes the form:

IF *expression* THEN *statement-a*;

Here, *statement-a* is executed if the value of expression has a rightmost bit of 1. Otherwise, nothing happens, and control immediately passes on to the next statement following the IF statement.

For example, the following sequence of PL/M statements will assign to INDEX either the number 5, or the value of THRESHOLD, whichever is larger. The value of INIT will change during execution of the IF statement only if THRESHOLD is greater than 5. The final value of INIT is copied to INDEX in any case:

```
INIT = 5;
IF THRESHOLD > INIT THEN INIT = THRESHOLD;
INDEX = INIT;
```

The power of the IF statement is enhanced by using DO blocks in the THEN and ELSE parts. Since a DO block can be used wherever a single statement can be used, each of the two statements in an IF statement may be a DO block. For example:

```
IF A = B THEN
            DO;
                 EQUAL$EVENTS = EQUAL$EVENTS + 1;
                 PAIR$VALUE  = A;
                 BOTTOM = B;
            END;
ELSE
            DO;
                 UNEQUAL$EVENTS = UNEQUAL$EVENTS + 1;
                 TOP = A;
                 BOTTOM = B;
            END;
```

DO blocks nested within an IF statement can contain further nested DO blocks, IF statements, variable and procedure declarations, and so on.

6.3.1 Nested IF Statements

Any IF statement (including the ELSE part, if any) can be considered a single PL/M statement (although it is not a block). Thus, the statement to be executed in a THEN or an ELSE clause may in fact be another IF statement.

An IF statement inside a THEN clause is called a nested IF. Nesting may be carried to several levels without needing to enclose any of the nested IF statements in DO blocks, as in the following construction:

```
IF expression-1 THEN
         IF expression-2 THEN
                   IF expression-3 THEN statement-a;
```

Here are three levels of nesting. Note that *statement-a* will be executed only if the values of all three expressions are true. Thus, the preceding example is equivalent to:

```
IF expression-1 AND expression-2 AND expression-3
THEN statement-a;
```

Notice that the preceding example of nesting does not have an ELSE part. When using nested IF statements, it is important to understand the following rule of PL/M:

- A set of nested IF statements can have only one ELSE part, and it belongs to the innermost (that is, the last) of the nested IF statements.

This rule could also be restated as follows:

- When an IF statement is nested within the THEN part of an outer IF statement, the outer IF statement may not have an ELSE part.

For example, the construction:

```
IF expression-1 THEN
   IF expression-2 THEN statement-a
   ELSE statement-b;
```

is legal and means that if the values of both *expression-1* and *expression-2* are true, then *statement-a* will be executed. If the value of *expression-1* is true and the value of *expression-2* is false, then *statement-b* will be executed. If the value of *expression-1* is false, neither *statement-a* nor *statement-b* will be executed, regardless of the value of *expression-2*.

The preceding construction is equivalent to:

```
IF expression-1 THEN
    DO;
         IF expression-2 THEN statement-a;
         ELSE statement-b;
    END;
```

This construction is much more readable and offers less opportunity for error.

If the intention is for the ELSE part to belong to the outer IF statement, then the nesting must be done by means of a DO block:

```
IF expression-1 THEN
    DO;
       IF expression-2 THEN statement-a;
    END;
ELSE statement-b;
```

Note that the meaning of this construction differs completely from the previous one.

Finally, consider the following:

```
   IF expression-1 THEN
          IF expression-2 THEN
                 IF expression-3 THEN statement-a;
                 ELSE statement-b;
          ELSE statement-c;          /* illegal statement */
   ELSE statement-d;                 /* illegal statement */
```

This construction is illegal because only one ELSE part is allowed. If the intention is for the ELSE parts to match the IF parts as indicated by the indenting, the nesting must be done with DO blocks, as follows:

```
IF expression-1 THEN
       DO;
              IF expression-2 THEN
              DO;
                             IF expression-3 THEN statement-a;
                             ELSE statement-b;
              END;
              ELSE statement-c;
       END;
ELSE statement-d;
```

6.3.2 Sequential IF Statements

Consider the following example. An ASCII-coded character is stored in a BYTE variable named CHAR. If the character is an A, *statement-a* should be executed. If the character is a B, *statement-b* should be executed. If the character is a C, *statement-c* should be executed. If the character is not A, B, or C, *statement-x* should be executed. The code for doing this could be written as follows, using IF statements that are completely independent of one another:

```
IF CHAR = 'A' THEN statement-a;
IF CHAR = 'B' THEN statement-b;
IF CHAR = 'C' THEN statement-c;
IF CHAR <> 'A' AND CHAR <> 'B' and CHAR <> 'C' THEN statement-x;
```

This sequence is inefficient because all four IF statements (six tests in all) will be carried out in every case, which is wasteful when one of the earlier tests succeeds.

A must be tested for in all cases. However, B needs to be tested only if the test for A fails and C needs to be tested only if both previous tests fail. Finally, if the tests for A, B, and C all fail, no further tests are needed and *statement-x* must be executed. To improve the code, rewrite it as follows:

```
IF CHAR = 'A' THEN statement-a;
ELSE IF CHAR = 'B' THEN statement-b;
ELSE IF CHAR = 'C' THEN statement-c;
ELSE statement-x;
```

Notice that this sequence is not a case of nested IF statements as described in the preceding section. IF statements are nested only when one IF statement is inside the THEN part of another. In the next example, IF statements are inside the ELSE parts of other IF statements. This construction is called sequential IF statements. It is equivalent to the following:

```
IF CHAR = 'A' THEN statement-a;
ELSE DO;
        IF CHAR = 'B' THEN statement-b;
        ELSE DO;
             IF CHAR = 'C' THEN statement-c;
             ELSE statement-x;
        END;
END;
```

Sequential IF statements are useful whenever a set of tests is to be made, but the remaining tests should be skipped whenever one of the tests succeeds. This construction works in such cases because all the remaining tests are in the ELSE part of the current test.

6.4 GOTO Statements

A GOTO statement alters the sequential order of program execution by transferring control directly to a labeled statement. Sequential execution then resumes, beginning with the target statement. The GOTO statement has the following form:

GOTO *label*

For example:

```
GOTO ABORT;
```

The appearance of *label* in a GOTO statement is called a label reference not a label definition.

The reserved word GOTO can also be written GO TO, with an embedded blank.

For reasons discussed in Chapter 7, GOTO statements are restricted. The only possible GOTO transfers are the following:

- From a GOTO statement in the outer level of some block to a labeled statement in the outer level of the same block.
- From a GOTO statement in an inner block to a labeled statement in the outer level of an enclosing block (not necessarily the smallest enclosing block). However, if the inner block is a procedure block, the transfer can only be to a statement in the outer level of the main program module.
- From any point in one program module to a labeled statement in the outer level of the main program module. To jump to such a label, the label must be declared to have extended scope, (i.e., declare it PUBLIC in the main module and EXTERNAL in the module containing the GOTO).

The use of GOTOs is necessary in some situations. However, in most situations where control transfers are desired, the use of an iterative DO, DO WHILE, DO CASE, IF, or a procedure activation (see Chapter 8) is preferable. Indiscriminate use of GOTOs will result in a program that is difficult to understand, correct, and maintain.

6.5 The CALL and RETURN Statements

The CALL and RETURN statements are mentioned here only for completeness, since they control the flow of a program. However, they are discussed in detail in Chapter 8.

The CALL statement is used to activate an untyped procedure (one that does not return a value).

The RETURN statement is used within a procedure body to cause a return of control from the procedure to the point from which it was activated.

Block Structure and Scope

7

Tabs for
452161-001

7 BLOCK STRUCTURE AND SCOPE CONTENTS

7 BLOCK STRUCTURE AND SCOPE

This chapter explains the meaning of outer level and the concept of scope, including the use of the linkage attributes, PUBLIC and EXTERNAL.

The outer level of a block means statements (or labels) contained in the block but not contained in any nested blocks. The term exclusive extent also has this meaning. The inner level, or inclusive extent, includes this outer level and all nested blocks as well.

A block at the same level as another block means that both blocks are contained by exactly the same outer blocks.

The scope of an object means those parts of a program where its name, type, and attributes are recognized (i.e., handled according to a given declaration). An object means a variable, label, procedure, or symbolic (named) constant (i.e., a compilation constant or execution constant as discussed in Chapter 3). A program is the complete set of modules that are ultimately executed as a unit.

These definitions are explained further in the following sections.

7.1 Names Recognized within Blocks

As shown throughout this manual, PL/M is a block-structured language that enables design implementation for problem solving, data processing, and hardware control.

PL/M is used to create blocks of code containing declarations followed by executable statements. These blocks are ordered and nested in such a way as to simplify and clarify the flow of data and control. (See Appendix B for maximum block nesting.) A collection of these blocks that performs a single function, or a small set of related functions, is usually compiled as one module, as discussed in Chapter 1.

Beyond the advantages of modularity, simplicity, and clarity, the nesting of blocks serves another very basic purpose: names declared at an outer level are known to all statements of all nested blocks as well.

A new meaning can be declared for any such name within a nested simple DO or procedure block, thereby cutting off its earlier meaning for this block. But if this option is not chosen, its meaning is established by a single declaration at an outer level. (The only objects that do not require declarations prior to use are labels and reentrant procedures.)

In Figure 7-1, everything inside the solid line constitutes the inclusive extent of block MMM (in this case, module MMM). KK is known throughout this block, including all nested blocks.

```
MMM: DO;                             /*Beginning of module*/
     DECLARE RECORD (128) STRUCTURE
             (KEY BYTE,
             INFO WORD);
     DECLARE CURRENT STRUCTURE
             (KEY BYTE,
             INFO WORD);
     DECLARE KK BYTE;
     KK = 127;
                 /* Instructions here would read in data. */
     SORT:  DO;
            DECLARE (JJ,II) INTEGER;
            DO JJ = 1 TO 127;
                    CURRENT.KEY = RECORD(JJ).KEY;
                    CURRENT.INFO = RECORD(JJ).INFO;
                    II = JJ;
            FIND:   DO WHILE II > 0 AND
                       RECORD(II-1).KEY > CURRENT.KEY;
                    RECORD(II).KEY = RECORD(II-1).KEY;
                    RECORD(II).INFO = RECORD(II-1).INFO;
                    II = II-1;
                    END FIND;
            RECORD(II).KEY = CURRENT.KEY;
            RECORD(II).INFO = CURRENT.INFO;
            END;
            END SORT;
     /* Instructions here would write out data from the records. */
END MMM;                                 /* End of module */
```

Figure 7-1 Inclusive Extent of Blocks

Everything inside the dashed line constitutes the inclusive extent of block SORT. JJ and II are known throughout this block, but not outside it. JJ and II are not known before the label SORT or after the END SORT statement.

Everything inside the dotted line constitutes the inclusive extent of block FIND. Since this is not a simple DO or procedure block, declarations are not allowed. All prior declarations shown are available for use within FIND.

In Figure 7-2, the shaded area is the exclusive extent (the outer level) of block SORT. The unshaded area within SORT is the exclusive (and inclusive) extent of block FIND. To the instructions within the FIND block, SORT's exclusive extent is an

```
MMM: DO;                                    /*Beginning of module*/
     DECLARE RECORD (128) STRUCTURE
             (KEY BYTE,
             INFO WORD);
     DECLARE CURRENT STRUCTURE
             (KEY BYTE,
             INFO WORD);
     DECLARE KK BYTE;
     KK = 127;
                /* Instructions here would read in data. */
     SORT:  DO;
            DECLARE (JJ,II) INTEGER;
            DO JJ = 1 TO 127;
                    CURRENT.KEY = RECORD(JJ).KEY;
                    CURRENT.INFO = RECORD(JJ).INFO;
                    II = JJ;
            FIND:   DO WHILE II > 0 AND
                       RECORD(II-1).KEY > CURRENT.KEY;
                    RECORD(II).KEY = RECORD(II-1).KEY;
                    RECORD(II).INFO = RECORD(II-1).INFO;
                    II = II-1;
                    END FIND;
            RECORD(II).KEY = CURRENT.KEY;
            RECORD(II).INFO = CURRENT.INFO;
            END;
            END SORT;

     /* Instructions here would write out data from the records. */
END MMM;                                /* End of module */
```

Figure 7-2 Outer Level of Block SORT

outer level. The outermost level (or module level) is the area outside the shaded area enclosing the SORT block.

7.1.1 Restrictions on Multiple Declarations

In any given block, a known name cannot be redeclared at the same level as its original declaration. A new declaration is permitted inside a nested simple DO or procedure block, where it automatically identifies a new object despite the existence of the same name at a higher level. The new object will be the only one known by this name within its block, and it will be unknown outside its block, where the prior name maintains its meaning. These observations also apply when a name is redeclared in another block at the same level as the block containing the original declaration.

When a name is declared only in a separate block at the same level, there is no way to access it except in that block where it is declared. The definition is not at an outer level to the current block. Any local declaration that is supplied establishes a new separate object whose values bear no relation to those of the other.

The reason for these rules, as for many in programming, is that there must be no ambiguity about what address/location is meant by each name in the program. The preceding declaration rules give freedom to choose names appropriate to a given block, without interfering with exterior uses of them. But when a name is redeclared, its outer-level meaning is inaccessible until execution exits the block containing the new declaration. For example:

```
A: DO;
     DECLARE X, Y, Z BYTE;
L1: X = 2;
     Y = X;
     Z = X;
B: DO;
     DECLARE X, Y BYTE;
     X = 3;
     Y = X;
L2: Z = X;
END B;
L3:     /* At this point, X=2, Y=2, Z=3, because the value of */
                           /* the redeclared X was used to fill Z */

   /* If statement L2 were outside the loop labeled B, then Z */
          /* would be 2 because the outer X value would be used */
```

7.2 Extended Scope: The PUBLIC and EXTERNAL Attributes

These attributes permit the scope of names to be extended for all objects except modules; a module name cannot be declared with either attribute.

To extend the scope means to make the names available for use in modules other than the one where they are defined. (The names are already available to nested blocks in this module.) To be specific, this includes names for variables, labels, procedures, and execution constants.

For example, the statement:

```
DECLARE FLAG BYTE PUBLIC;
```

causes a byte named FLAG to be allocated, and its address made known to any other module where the following declaration occurs:

```
DECLARE FLAG BYTE EXTERNAL;
```

Similarly, if one module has a procedure declaration block that begins:

```
SUMMER: PROCEDURE (A,B) WORD PUBLIC;
        DECLARE (A,B) BYTE;
        .           /* other declarations can go here */
        .           /* executable statements go here, */
        .                   /* defining the procedure */
        .
END SUMMER;
```

then any other module may invoke SUMMER if it first declares:

```
SUMMER: PROCEDURE (A,B) WORD EXTERNAL;     /* A,B can be any names */
        DECLARE (A,B) BYTE;     /* but these names must match them */
               /* and each type must match its public definition */
END SUMMER;
```

The use of PUBLIC and EXTERNAL must follow a strict set of rules to prevent ambiguity of location or definition. These rules are as follows:

1. These attributes can be used only in a declaration at the outermost level of a module (i.e., never in a nested block).
2. Only one can appear in any declaration, no more than once. Thus:

   ```
   DECLARE ZETA BYTE PUBLIC EXTERNAL;                /* error */
   DECLARE RHO WORD PUBLIC PUBLIC;                   /* error */
   ```

   and similar constructs are all invalid.

3. Names can be declared PUBLIC, no more than once. The PUBLIC declaration is the defining declaration: the address it creates is used in each procedure or module where the same name is declared EXTERNAL. Do not create more than one PUBLIC address for any name.
4. Names can be declared EXTERNAL only if they are also declared PUBLIC in a different module of the program. The EXTERNAL attribute is essentially a request to use a PUBLIC address. An EXTERNAL without a PUBLIC is a dead letter. Lack of a definition elsewhere will result in a link-time error.
5. Where the name is declared EXTERNAL, it must be given the same type as where it is declared PUBLIC. Any contradiction of type would violate the intention to use the location(s) and content(s) defined elsewhere. If the name is declared PUBLIC and has the DATA attribute, all EXTERNAL declarations must also use DATA, but cannot assign a value to the constant being declared.
6. Similarly, names declared EXTERNAL must not be given a location (using the AT clause), or an initialization (using DATA or INITIAL). Such usage would contradict the fact that names are being defined in another module. However, in the module where this name is declared PUBLIC, the use of AT, DATA (with initialization values present), or INITIAL is allowed.
7. Neither PUBLIC nor EXTERNAL can be applied to a name that is based. For example:

```
DECLARE PTR1 POINTER;
DECLARE V1 BASED PTR1 PUBLIC;
```

   is invalid. The reason: by definition, V1 has no home of its own; its location is always determined by PTR1. Thus, to declare V1 PUBLIC or EXTERNAL does not permit the correct assignment of addresses. PTR1, on the other hand, always contains the current address of V1. Declaring the base, in this case PTR1, to be PUBLIC or EXTERNAL is always permissible since it permits valid results.

NOTE

The PL/M compiler will generate external records only for items that are actually referenced in the program.

8. When extending the scope of a name with the PUBLIC attribute and DATA or INITIAL, the placement in the DECLARE statement is critical. PUBLIC must be placed after the type declaration and before the DATA or INITIAL attribute. For example:

```
DECLARE a$p BYTE PUBLIC INITIAL(4);
```

(Additional restrictions on the use of PUBLIC and EXTERNAL procedures are described in Chapter 8.)

Following these rules will enable consistent and reliable execution of programs using names with extended scope. A PUBLIC definition occurring in one module will then be used by all related references to that name in separate modules; that is, references which declare the name EXTERNAL. The following diagram illustrates this:

```
 MOD1: DO;
         DECLARE V1 BYTE PUBLIC;
           .
           .
           .
       END MOD1;
```

```
 MOD2: DO;
         DECLARE V1 BYTE EXTERNAL;
         QQ4: PROCEDURE PUBLIC;
           .
           .
           .
         END QQ4;
       END MOD2;
```

Both references to V1 will use the same definition (location) for V1, namely, the definition in module MOD1. Similarly, if any module needed to call procedure QQ4, it would first need a declaration like this:

```
QQ4:PROCEDURE EXTERNAL;
END QQ4;
```

so that a subsequent CALL QQ4 would correctly pass control to that procedure in MOD2.

7.3 Scope of Labels and Restrictions on GOTOs

Labels are subject to exactly the same rules of scope previously discussed.

A label is unknown outside the block where it is declared. As discussed in Chapter 1, a label is either declared explicitly at the beginning of a simple-DO or procedure block, or the compiler considers it to be declared there as soon as it is defined by use anywhere in the block. Therefore, the discussion of what names are known in which blocks applies directly to labels as well as to other names.

The label on a block is not part of the block it names. For example, the name on the DO enclosing the module itself is not part of the DO; it merely names it. For nested blocks, a label is again not part of the block it names, but belongs instead to the outer level as part of that first enclosing block.

If a name used as a label on a block is defined inside that block, it will name a new item, be it label, variable, or constant. There will be no confusion with the outer label name. This fact leads to important restrictions on the use of the GOTO statement:

1. It is impossible for a GOTO to transfer control from an outer block to a labeled statement inside a nested block.
2. Moreover, a GOTO can transfer control from one block to another in the same module only if the target block enclosed the one containing the GOTO (and only if the name of that target label is not declared in the nested block).

Furthermore, a label with the PUBLIC attribute is permitted only in the main module. This has the interesting consequence of forcing all other transfers of control (i.e., those not involving a return to the main module) to use procedure calls. This favors the development of orderly, modularized, traceable programs.

Only four GOTO transfers are possible; these are as follows:

1. From one point in a block to another statement also in the same level of the same block.
2. From an inner, nested DO-block (not a nested procedure) to a statement in the outer level of any enclosing block.
3. From a procedure to a statement in the outer level of the main program in the same module.
4. To a main-program label that is declared PUBLIC, from any point in any module that declares that label EXTERNAL.

(Recall that only labels at the outer level of a main program can be declared PUBLIC.)

Given the program structure and declarations shown in Figure 7-3, the only valid GOTO transfers are shown in Figure 7-4. A single-headed arrow means the transfer is valid only in the direction shown. A double-headed arrow means that a GOTO can be used in either direction.

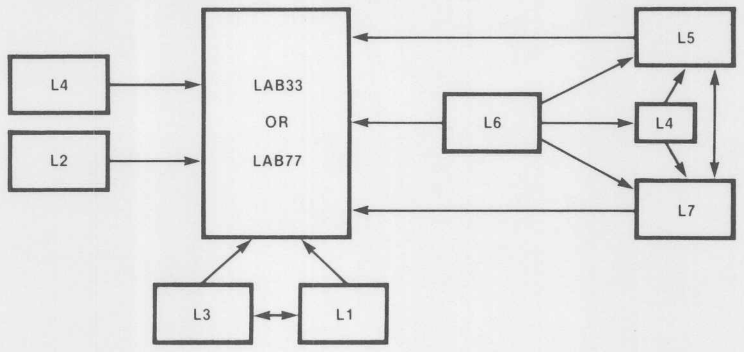
Figure 7-4 illustrates legal GOTO transfers that are the only transfers permitted among the given labels in Figure 7-3.

```
MAIN: DO;
      DECLARE (LAB33, LAB77) LABEL PUBLIC;
      DECLARE IT BYTE;
      .
      .
      .
    LAB33: . . . ;
               DO;
               .
               .
               .
               END;
      .
      .
      .
    LAB77: . . . ;
               DO WHILE IT > 0;
               .
               .
               .
               END;
      .
      .
      .
      END MAIN;
MOD1: DO;
      DECLARE (LAB33,LAB77) LABEL EXTERNAL;
      P1: PROCEDURE;
            L1: . . . ;
                    DO;
                    DECLARE KO BYTE;
                    P2: PROCEDURE;
                            .
                            .
                            .
                        L2: . . .;
                            .
                            .
                            .
                     END P2;
                     END;
            L3: . . . ;
                    .
                    .
                    .
            END P1;
END MOD1;
```

Figure 7-3 Sample Program Modules Illustrating Valid GOTO Usage

```
MOD2: DO;
      .
      .
      .
      DECLARE (LAB33,LAB77) LABEL EXTERNAL;
      P4: PROCEDURE;
      .
      .
      .
      L4: . . . ;
      .
      .
      .
      L5: . . . ;
              DO;
                  L6: . . . ;
                      .
                      .
                      .
              END;
      .
      .
      .
      L7: . . . ;
      .
      .
      .
  END P4;
  L8: . . . ;
END MOD2;
```

Figure 7-3 Sample Program Modules Illustrating Valid GOTO Usage (continued)



121636-1

Figure 7-4 Sample Program Modules Illustrating Valid GOTO Transfers

Procedures 8

Tabs for
452161-001

8 PROCEDURES CONTENTS

8 PROCEDURES

A procedure is a section of PL/M code that is declared without being executed, and then activated from other parts of the program. A function reference or CALL statement activates the procedure, even if it is physically located elsewhere. Program control is transferred from the point of activation to the beginning of the procedure code, and the code is executed. Upon exit from the procedure code, program control is passed back to the statement immediately after the point of activation.

The use of procedures forms the basis of modular programming. It facilitates making and using program libraries, eases programming and documentation, and it reduces the amount of object code generated by a program. The following sections review how to declare and activate procedures.

8.1 Procedure Declarations

A procedure must be declared, just as variables must be declared. Thereafter, any reference to a procedure must occur within the scope defined by the procedure declaration. Also, a procedure cannot be used (called, or invoked in an expression) until after the END statement of the procedure declaration (unless reentrant, see Section 8.5).

A procedure declaration consists of three parts: a PROCEDURE statement, a sequence of statements forming the procedure body, and an END statement.

The following is a simple example:

```
DOOR$CHECK: PROCEDURE;
                IF FRONT$DOOR$LOCKED AND SIDE$DOOR$LOCKED THEN
                    CALL POWER$ON;
                ELSE CALL DOOR$ALARM;
    END DOOR$CHECK;
```

where POWER$ON and DOOR$ALARM are procedures declared previously in the same program.

NOTE

The name in a PROCEDURE statement has the same appearance as a label definition, but it is not considered a label definition, and a procedure name is not a label. PROCEDURE statements cannot be labeled.

The name is a PL/M identifier, which is associated with this procedure. The scope of a procedure is governed by the placement of its declaration in the program text, just as the scope of a variable is governed by the placement of its DECLARE statement (see Chapter 7 for a detailed description). Within this scope, the procedure can be activated by the name used in the PROCEDURE statement.

A procedure declaration, like a DO block, controls the scope of variables as described in Chapter 7. Also, like a simple DO block, a procedure declaration can contain DECLARE statements, which must precede the first executable statement in the procedure body.

As in a DO block, the identifier in the END statement has no effect on the program, but helps legibility and debugging. If used, it should be the same as the procedure name.

The parameter list and the type are discussed in the following two sections.

8.1.1 Parameters

Formal parameters are non-based scalar variables declared within a procedure declaration; their identifiers appear in the parameter list in the PROCEDURE statement. The identifiers in the list are separated by commas and the list is enclosed in parentheses. No subscripts or member-identifiers can be used in the parameter list.

If the procedure has no formal parameters, the parameter list (including the parentheses) is omitted from the PROCEDURE statement.

Each formal parameter must be declared as a non-based scalar variable in a DECLARE statement preceding the first executable statement in the procedure body. However, procedure parameters are not stored according to the same rules as other declared variables. In particular, do not assume that a parameter is stored contiguously with other variables declared in the same factored variable declaration.

When a procedure that has formal parameters is activated, the CALL statement or function reference contains a list of actual parameters. Each actual parameter is an expression whose value is assigned to the corresponding formal parameter in the procedure before the procedure begins to execute.

For example, the following procedure takes four parameters, called PTR, N, LOWER, and UPPER. It examines N contiguously stored BYTE variables. The parameter PTR is the location of the first of these variables. If any of these variables is less than the parameter LOWER or greater than the parameter UPPER, the ERRORSET procedure (declared previously in the program) is activated:

```
RANGE$CHECK: PROCEDURE(PTR, N, LOWER, UPPER);
     DECLARE PTR POINTER;
     DECLARE (N, LOWER, UPPER, I) BYTE;
     DECLARE ITEM BASED PTR(1) BYTE;

     DO I = 0 TO N - 1;
        IF (ITEM(I) < LOWER) OR (ITEM(I) > UPPER)
        THEN CALL ERRORSET;

        /* ERRORSET is a procedure declared previously */

     END;
END RANGE$CHECK;
```

Notice that the array ITEM is declared to have only one element. Since it is a based array, a reference to any element of ITEM is really a reference to some location relative to the location represented by PTR. In writing the procedure RANGE $CHECK, a dimension specifier that is any arbitrary number greater than zero must be supplied for ITEM so that references to ITEM can be subscripted. But it does not matter what the dimension specifier is (1 is arbitrarily used here).

Having made this declaration, suppose that 25 variables are stored contiguously in an array called QUANTS. To check that all of these variables have values within the range defined by the values of two other BYTE variables, SMALL and LARGE, write:

```
CALL RANGE$CHECK (@QUANTS, 25, SMALL, LARGE);
```

When this CALL statement is processed, the following sequence occurs:

- The four actual parameters in the CALL statement (@QUANTS, 25, SMALL, and LARGE) are assigned to the formal parameters PTR, N, LOWER, and UPPER, which were declared within the procedure RANGE$CHECK. Since ITEM is based on PTR and the value of PTR is @QUANTS, every reference to an element of ITEM becomes a reference to the corresponding element of QUANTS.

- The executable statements of the procedure RANGE$CHECK are executed. If any of the values are less than the value of SMALL or greater than the value of LARGE, the procedure ERRORSET is activated.
- Finally, control returns to the statement following the CALL statement.

Notice how the use of a based variable, with the base passed as a parameter, allows the procedure to have its own unchanging name (ITEM) for a set of variables which may be a different set each time the procedure is activated.

Parameters are placed on the stack in left-to-right order. The stack grows from higher locations to lower locations, so the first parameter occupies the highest position on the stack, and the last parameter occupies the lowest position. For more information, see Appendix F.

NOTE

PL/M does not guarantee the order in which multiple actual parameters will be evaluated when the procedure is activated. If one actual parameter changes another actual parameter, the results are undefined. This can occur if an expression used as an actual parameter contains an embedded assignment or function reference that changes another actual parameter for the same procedure.

8.1.2 Typed Versus Untyped Procedures

The preceding RANGE$CHECK procedure is an untyped procedure. No type is given in the PROCEDURE statement, and it does not return a value. An untyped procedure is activated by using its name in a CALL statement, as explained in Section 8.2.

A typed procedure, also called a function, has a type in its PROCEDURE statement: an unsigned binary number, signed integer, real number, pointer or selector data type. Such a procedure returns a value of this type, which is used in an expression or stored as the value of a variable. The procedure is activated by using its name as an operand in an expression; this special type of variable reference is called a function reference.

When the expression is processed at run time, the function reference causes the procedure to be executed. The function reference itself is then replaced by the value returned by the procedure. The expression containing the function reference is then evaluated, and program execution continues in normal sequence.

Like an untyped procedure, a typed procedure can have parameters. They are handled as described in the previous section.

NOTE

The body of a typed procedure can contain a RETURN statement with an expression, as explained later in this chapter.

The body of a typed procedure can contain code (such as an assignment statement) that changes the value of some variable declared outside the procedure. This is called a side effect.

Recall that PL/M does not guarantee the order in which operands in an expression are evaluated. Therefore, if a function used in an expression changes the value of another variable in the same expression, the value of the expression depends on whether the function reference or the variable is evaluated first.

If the analysis of the expression does not force one of these operands to be evaluated before the other, then the value of the expression is undefined.

This situation can be avoided by using parentheses to segregate any typed procedure that has a side effect, or by using this procedure in an assignment statement first to create an unambiguous sequence.

8.2 Activating a Procedure — Function References and CALL Statements

The two forms of procedure activation depend on whether the procedure is typed or untyped. An untyped procedure is activated by means of a CALL statement, which has the form:

```
CALL name;
```

or

```
CALL name (parameter list);
```

For example:

```
CALL REORDER (@RANK$TABLE,3);
```

(An alternate form of the CALL statement is discussed later.)

A typed procedure is activated by means of a function reference, which is an operand in an expression. A function reference has the form:

```
name
```

or

```
name(parameter list)
```

This occurs as an operand in an expression, as in the following example:

```
TOTAL = SUBTOTAL + SUM$ARRAY (@ITEMS,COUNT);
```

where SUM$ARRAY is a previously declared typed procedure. The value added to SUBTOTAL will be the value returned by SUM$ARRAY using the actual parameters (@ITEMS, COUNT).

In both forms of procedure activation, the elements of the parameter list are called actual parameters to distinguish them from the formal parameters of the procedure declaration. At the time of activation, each actual parameter is evaluated and the result is assigned to the corresponding formal parameter in the procedure declaration. Then, the procedure body is executed. Any PL/M expression may be an actual parameter if its type is the same as that of the corresponding formal parameter.

The actual parameter list in a procedure activation must also match the formal parameter list in the procedure declaration. That is, it must contain the same number of parameters of the same type (except as described in the next paragraph) in the same order. If the procedure is declared without a formal parameter list, then no actual parameter list can be used in the activation.

As in expression evaluation and assignment statements (see Chapter 5), a few type conversions are performed automatically when necessary in activating and returning from a procedure. The built-in explicit type conversion procedures described in Chapter 9 can also be used to force the value of an expression to a desired type.

8.2.1 Indirect Procedure Activation

The CALL statement, in the form shown in the preceding section, activates an untyped procedure by its name. It is also possible to activate an untyped procedure by its location. This is done by means of a CALL statement with the form:

CALL *identifier*[.*member-identifier*] [(*parameter list*)];

The identifier cannot be subscripted; however it can be a structure reference. The identifier must be a fully qualified POINTER or WORD type variable reference for PL/M-86 and PL/M-286, and a fully qualified POINTER, OFFSET, or WORD type variable reference for PL/M-386. Its value is assumed to be the location of the entry-point of the procedure being activated.

PL/M-86/286

NOTE

Calls through 32-bit POINTERs will be translated into long calls whereas calls through 16-bit WORDs or POINTERs (in the SMALL case) will be translated into short calls (relative to the current code segment). The compiler will issue a warning if the wrong addressing type is used to gain a procedure address for later indirect calls.

end PL/M-86/286

PL/M-386

NOTE

Calls through 48-bit POINTERs will be translated into long calls whereas calls through 32-bit OFFSETs, WORDs, or POINTERs (in the SMALL case) will be translated into short calls (relative to the current code segment).

The identifier for the indirect procedure activation cannot be an HWORD. Therefore, all variables used for indirect calling in programs that are recompiled from PL/M-286 and use the WORD16 control should have DWORD, OFFSET (or ADDRESS) data types.

end PL/M-386

A normal CALL uses the name of the procedure; the compiler checks to make sure that the correct number of parameters is supplied and performs automatic type conversion on the actual parameters.

When the CALL statement uses a location, the compiler does not check the number of parameters or perform type conversion. However, type conversion is performed if the actual argument is a constant expression. The constant expression is evaluated in unsigned context, as described in Section 5.6. If the number of parameters is wrong or if an actual parameter is not of the same type as the corresponding formal parameter, the results are unpredictable.

8.3 Exit from a Procedure: The RETURN Statement

The execution of a procedure is terminated in one of three ways:

- By execution of a RETURN statement within the procedure body. A typed procedure must terminate with a RETURN statement that has an expression.
- By executing a GOTO to a statement outside the procedure body. The target of the GOTO must be at the outer level of the main program (see Chapter 7).
- By reaching the END statement that terminates the procedure declaration.

The RETURN statement takes one of two forms:

```
RETURN;
```

or

```
RETURN expression;
```

The first form is used in an untyped procedure. The second form is used in a typed procedure. The value of the expression becomes the value returned by the procedure. It is evaluated as if it were being assigned to a variable of the same type as used on the PROCEDURE statement.

8.4 The Procedure Body

The statements within the procedure body can be any valid PL/M statements, including CALL statements as well as nested procedure declarations.

8.4.1 Examples

1. The following is a typed procedure declaration:

```
AVG: PROCEDURE (X,Y) REAL;
          DECLARE (X,Y) REAL;
          RETURN (X + Y)/2.0;
END AVG;
```

This procedure could be used as follows:

```
SMALL = 3.0;
LARGE = 4.0;
MEAN = AVG (SMALL, LARGE);
```

The effect would be to assign the value 3.5 to MEAN.

2. The following is an untyped procedure:

```
AOUT: PROCEDURE (ITEM);
        DECLARE ITEM WORD;
        IF ITEM >= 0FFH THEN COUNTER = COUNTER + 1;
        RETURN;
END AOUT;
```

Here COUNTER is some variable declared outside the procedure (i.e., it is a global variable). This procedure could be activated as follows:

```
CALL AOUT (UNKNOWN);
```

If the value of the variable UNKNOWN is greater than or equal to 0FFH, the value of COUNTER will be incremented.

3. This example demonstrates an important use of based variables:

```
SUM$ARRAY: PROCEDURE (PTR,N) BYTE;
        DECLARE PTR POINTER,
                     ARRAY BASED PTR(1) BYTE, (N,SUM,I)BYTE;
        SUM = 0;
        DO I = 0 TO N;
            SUM = SUM + ARRAY(I);
        END;
        RETURN SUM;
END SUM$ARRAY;
```

This procedure returns the sum of the first N + 1 elements (from the zeroth to the Nth) of a BYTE array pointed to by PTR. Notice that ARRAY is declared to have 1 element. Since it is a based variable, no space is allocated for it. It must be declared as an array (with a non-zero dimension) so that it can be subscripted in the iterative DO block. The choice of 1 as the constant in the dimension specifier is arbitrary and does not restrict the value of N that may be supplied when the procedure is activated.

The procedure could be used as follows to sum the elements of a 100-element BYTE array named PRICE, and to assign the sum to the variable TOTAL:

```
TOTAL = SUM$ARRAY(@PRICE,99);
```

8.5 The Attributes: PUBLIC and EXTERNAL, INTERRUPT, REENTRANT

The PUBLIC and EXTERNAL attributes can be included in PROCEDURE statements to give procedures extended scope. Extended scope is discussed in Chapter 7.

A procedure declaration with the PUBLIC attribute is called a defining declaration. A procedure declaration with the EXTERNAL attribute is called a usage declaration. Most of the rules for PUBLIC and EXTERNAL appear in Chapter 7. The following additional rules apply to the use of the EXTERNAL attribute in a procedure declaration:

1. The EXTERNAL attribute cannot be used in the same PROCEDURE statement as a PUBLIC or REENTRANT attribute. Note, however, that the defining declaration of a procedure may have the REENTRANT attribute.

2. A usage (EXTERNAL) declaration of a procedure should have the same number of parameters as the defining (PUBLIC) declaration. Variable types and dimension specifiers should match up in the same sequence in both declarations. The names of the parameters need not be the same. Note that a discrepancy between the parameter lists in the defining declaration and in a usage declaration will not be automatically detected. (See Chapter 11 for a description of the TYPE control to detect such an error at module linkage time.)

3. The procedure body of a usage declaration cannot contain anything except the declarations of the formal parameters. The formal parameters must be declared with the same types as in the defining declaration.

4. No labels can appear in a usage declaration.

NOTE

The PL/M compiler will generate external records only for items that are actually referenced in the program.

For example, the procedure AVG (from example 1 in Section 8.4.1) can be altered by giving it the PUBLIC attribute:

```
AVG: PROCEDURE (X,Y) REAL PUBLIC;
          DECLARE (X,Y) REAL;
          RETURN (X + Y)/2.0;
END AVG;
```

Another module would have a usage declaration, as follows:

```
AVG: PROCEDURE (X,Y) REAL EXTERNAL;
         DECLARE (X,Y) REAL;
END AVG;
```

Now, in the module with the usage declaration, AVG can be referenced in an executable statement:

```
MIDDLE = AVG (FIRST, LATEST);
```

thereby activating the procedure AVG as declared in the first module.

8.5.1 Interrupts and the INTERRUPT Attribute

The INTERRUPT attribute enables definition of a procedure to handle some condition signaled by a microprocessor interrupt (e.g., from a peripheral device). A procedure with this attribute is activated when the corresponding interrupt signal is received in the target system. The PL/M statement CAUSE$INTERRUPT (constant) can also be used to initiate an interrupt signal (see Chapter 10).

Note that the following discussion applies only to interrupt procedures; interrupt tasks are discussed in Appendix G.

PL/M-86

The INTERRUPT attribute can be used only in declaring an untyped procedure with no parameter at the outermost level of a program module. It must be declared PUBLIC or EXTERNAL (and optionally REENTRANT). The form is:

```
INTERRUPT n
```

where *n* is any decimal number from 0 to 255. Each number can be used only once in a program. Each procedure is referred to as an interrupt procedure. Such a procedure is necessary to provide non-default handling of exception conditions arising in floating-point arithmetic (see Appendixes F and G).

end PL/M-86

PL/M-286/386

The INTERRUPT attribute can be used only in declaring an untyped procedure with no parameters at the outermost level of a program module. It must be declared PUBLIC or EXTERNAL (and optionally REENTRANT). The form is:

```
INTERRUPT
```

At build time, an interrupt vector is assigned to each interrupt procedure.

end PL/M-286/386

The following discussion of the microprocessor interrupt mechanism clarifies how interrupt procedures work. Additional information can be found in Appendix G.

The microprocessor interrupt mechanism has two states: enabled or disabled. With the ENABLE statement, interrupts can take effect. The DISABLE statement prevents interrupts from having any effect. The HALT statement also enables interrupts. (The state of the microprocessor interrupt mechanism upon initialization is determined by the operating system.)

When some peripheral device sends an interrupt to the microprocessor CPU, it is ignored if the interrupt mechanism is disabled. If interrupts are enabled, the interrupt is processed as follows:

1. The CPU completes any instruction currently in execution.
2. The CPU sends an acknowledge interrupt signal, then the interrupting device sends its interrupt number.
3. The interrupt mechanism is disabled. This prevents any other device from interfering.
4. Control passes to the interrupt procedure whose number matches the number sent by the peripheral device. If no such procedure has been established, the results are undefined (since the vector that transfers control may be uninitialized).
5. When the procedure is through (by executing a RETURN or reaching the END of the procedure), the interrupt mechanism is enabled so other devices can be serviced, and control returns to the point where the interrupt occurred.

   It is possible (as with other untyped procedures) for the procedure to terminate by executing a GOTO with a target outside the procedure in the outer level of the main program module. In this case, control will never be returned to the point where the program was interrupted, and interrupts will not be enabled automatically.

The following is an example of an interrupt procedure for a system where a peripheral device initiates an interrupt whenever the temperature of a device exceeds a certain threshold. The interrupt procedure turns on the annunciator light, updates a status word, and returns control to the program:

```
HITEMP: PROCEDURE INTERRUPT 100 PUBLIC;
        CALL ANNUNCIATOR(1);
            /* This will result in an output from the microprocessor
                          to turn on annunciator light number 1, the
                                        high-temperature warning. */
        ALERT = ALERT OR 00000010B;
              /* This puts a 1 in one of the bit positions of ALERT,
        which contains a bit pattern representing current alerts. */
END HITEMP;
```

8.5.2 Activating an Interrupt Procedure with a CALL Statement

A procedure with the INTERRUPT attribute can also be activated by a CALL statement. However, interrupts are not automatically disabled on activation of the procedure. If interrupts are enabled when the CALL is executed, then unless the procedure has a DISABLE statement as its first executable statement, it will run with interrupts enabled and should have the REENTRANT attribute (see next section).

NOTE

If you use an interrupt procedure that will call another interrupt procedure, the length of the stack storage must be extended manually, since the compiler allocates only enough space to save one procedure. See Appendix G and use the program combining sequence.

An interrupt procedure activated by a CALL statement is like any other procedure so activated.

NOTE

Unlike PL/M-80, PL/M-86 interrupt routines activated with a CALL statement do not alter the interrupt enable status. This means that termination of the procedure by means of a RETURN statement or the END statement will not automatically enable interrupts.

Section 9.7.2 describes the built-in function INTERRUPT$PTR, which returns the interrupt entry point, given an interrupt procedure name. Section 9.7.1 also describes the built-in procedure SET$INTERRUPT, which sets an interrupt vector given the interrupt procedure name and number.

The CAUSE$INTERRUPT statement causes a software interrupt to the vector specified in the statement:

```
CAUSE$INTERRUPT(constant);
```

where *constant* is in the range 0 to 255.

end PL/M-86

8.5.3 Reentrancy and the REENTRANT Attribute

With the REENTRANT attribute, a procedure can suspend execution temporarily, restart with new parameters, and then later complete the original execution successfully as if there had been no interruption.

This ability is desirable in two circumstances: (1) if the procedure (PROC1) activates itself (called direct recursion), or (2) if the procedure activates another procedure (PROC2) that will reactivate PROC1 before PROC1 has finished its original processing (called indirect recursion).

Without the REENTRANT attribute, storage for procedure variables is allocated statically, in fixed locations within the data segment of the object module. Reentering such a procedure would write over the earlier contents of such locations making it impossible to complete the original suspended execution.

When the attribute REENTRANT is used in declaring a procedure, its variables are not stored with other variables in the data section, but are stored on the stack. Thus preserved, each set can be used independently by each invocation of the procedure.

Hence, multiple sets of variables might need to be stored on the stack during recursive use of such procedures. A stack size must be specified (when combining the program module) that is large enough for all such storage needed by all multiple invocations that may be active at one time.

A procedure with the REENTRANT attribute may be activated before it is declared. This permits direct recursion, where the procedure activates itself and permits indirect recursion, where the procedure activates a second procedure and the second procedure activates the first, or activates a third procedure, which activates a fourth, and so forth, with the result that the first procedure is activated before it terminates.

The following rules summarize the use of the REENTRANT attribute:

- Any procedure that can be interrupted and is also activated from within an interrupt procedure should have the REENTRANT attribute.

  Note that this may apply to an interrupt procedure that runs with interrupts enabled because it contains an ENABLE statement. If there is any possibility that it will be interrupted by its own interrupt, it should have the REENTRANT attribute. This situation is equivalent to recursion.

- Any procedure that is directly recursive (activates itself) should have the REENTRANT attribute.

- Any procedure that is indirectly recursive (activates another procedure and is activated itself as a result) should have the REENTRANT attribute.
- Any procedure that is activated by a reentrant procedure should also have the REENTRANT attribute. In other words, if there is any possibility that a procedure can be activated while it is already running, it should be REENTRANT.
- The REENTRANT attribute cannot be used in the same declaration as the EXTERNAL attribute. (It may be used with the PUBLIC attribute.)
- The REENTRANT attribute can only be used in a PROCEDURE statement at the outer level of a module.
- A procedure declaration with the REENTRANT attribute cannot have a nested procedure declaration.

Built-in Procedures,
Functions, and Variables 9

Tabs for
452161-001

9 BUILT-IN PROCEDURES, FUNCTIONS, AND VARIABLES CONTENTS

intel

9 BUILT-IN PROCEDURES, FUNCTIONS, AND VARIABLES

Built-in procedures, functions, and variables are already declared in the PL/M code. This makes it unnecessary to write code to perform the particular functions that built-ins are designed to perform. The following built-in procedures, functions, and variables are discussed in this chapter:

- LENGTH/LAST/SIZE functions — these functions return information concerning variables. For example, the SIZE function returns the number of bytes occupied by a scalar, array, or structure.
- Explicit type and value conversion functions — these functions provide explicit conversion for types and values.
- Shift and rotate functions — these functions move bits using a pattern of 8, 16, or 32 bits.
- String manipulation procedures and functions — these procedures and functions move strings, compare strings, search strings for a match or a mismatch, translate strings, and set strings to a specified value.
- Bit manipulation procedures (specific to PL/M-386) — these functions copy (and move) a bit string and search bit strings for a set bit.
- MOVE bytes — this procedure moves a specified number of bytes from one location to another.
- Time delay — this procedure causes a time delay.
- Lock set — this function enables a software synchronization lock.
- Lock bit — this function enables a memory location lock.
- Interrupt procedures (specific to PL/M-86) — these procedures enable the setting of an interrupt vector and an interrupt-handling procedure entry point.
- POINTER and SELECTOR functions — these functions enable the manipulation of location addresses in the microprocessor's memory.

The identifiers for these built-ins are subject to the rules of scope (described in Chapter 7). This means that the name of a built-in procedure or variable can be declared to have a local meaning (scope) within the program. Within the scope of such a declaration, the built-in is unavailable. This distinguishes these identifiers from reserved words (listed in Appendix A), which cannot be used as identifiers in declarations.

No built-in procedure can be used within a location reference (e.g., @LENGTH(LIST)). No built-in variable can be used within a location reference, except as specifically noted in the following sections.

9.1 Obtaining Information About Variables

PL/M has three built-in procedures that take variable names as actual parameters and return information based on the declarations of the variables: LENGTH, LAST, and SIZE.

9.1.1 The LENGTH Function

LENGTH is a built-in WORD function that returns the number of elements in an array; it is activated by a function reference with the form:

LENGTH (*variable-ref*)

Where:

variable-ref	must be a non-subscripted reference to an array.

The array can be a member of a structure; it cannot be an EXTERNAL array using the implicit dimension specifier (see Section 3.2.1).

The value returned is the number of elements assigned to the array in the declaration statement (i.e., the value of the dimension specifier).

If the array is not a structure member, then the reference must be an unqualified variable reference. If the array is a structure member, then the reference is a partially qualified variable reference. For example, given the declaration:

```
DECLARE RECORD STRUCTURE (KEY BYTE,
                           INFO(3) WORD);
```

LENGTH(RECORD.INFO) is a valid function reference and returns a WORD value of 3.

If the array is a member of a structure, and that structure is an element of an array, a special case arises. Given the declaration:

```
    DECLARE LIST (4) STRUCTURE (KEY BYTE,
                                INFO (3) WORD);
```

then all of the following function references are correct and return the value 3:

```
    LENGTH(LIST(0).INFO)
    LENGTH(LIST(1).INFO)
    LENGTH(LIST(2).INFO)
    LENGTH(LIST(3).INFO)
```

In other words, the subscript for the array LIST is irrelevant when a member-identifier is supplied, because the arrays within the structures are all the same length. PL/M has a shorthand form of partially qualified variable reference in the LENGTH, LAST, and SIZE function references. For example:

```
    LENGTH(LIST.INFO)
```

is a valid function and returns the value 3.

9.1.2 The LAST Function

LAST is a built-in WORD function that returns the subscript of the last element in an array. It is activated by a function reference with the form:

LAST (*variable*)

Where:

variable	must be a non-subscripted reference to an array.

The array can be a member of a structure; it cannot be an EXTERNAL array using the implicit dimension specifier (see Section 3.2.1).

The value returned is the subscript of the last element of the array. For a given array, LAST will always be one less than LENGTH. When used with a based variable, LAST returns the value assigned in the declaration statement. This is not necessarily the actual value.

As in the LENGTH function, a shorthand form of partially qualified variable reference is allowed in the case where the array is a member of a structure that is also an array element.

9.1.3 The SIZE Function

SIZE is a built-in WORD function that returns the number of bytes occupied by a scalar, array or structure. It is activated by a function reference with the form:

SIZE (*variable*)

Where:

variable	is a fully qualified, partially qualified, or unqualified reference to any scalar, array, or structure. The variable cannot be an EXTERNAL declaration that uses the implicit dimension specifier (see Section 3.2).

The value returned is the number of bytes required by the variable referenced. When used with a based variable, SIZE returns the value assigned in the declaration statement. This is not necessarily the actual (current) value.

If the reference is fully qualified, it refers to a scalar, and the value is the number of bytes required for the scalar. If the reference is unqualified, it refers to an entire structure or array, and the value is the total number of bytes required for the structure or array.

If the reference is partially qualified, it refers either to a structure member that is an array or nested structure, or to an array element that is a structure. The value is the number of bytes required for the array or structure.

As in the LENGTH function, a shorthand form of partially qualified variable reference is allowed in the case where the array or scalar is a member of a structure and the structure is an array element.

9.2 Explicit Type and Value Conversions

The functions in this section provide explicit conversion from one data type to another and from signed values to or from absolute magnitudes.

Explicit type and value conversion functions are invoked as:

function-name (*expression*)

In Tables 9-1 and 9-2, each function name is followed by the expression type expected, the purpose of the function, and the nature of the value it returns to the expression that invoked it. For each function there is only one possible class of expressions (e.g., HIGH accepts only unsigned values) that can be converted. For the type conversions (BYTE, WORD, DWORD, INTEGER, REAL, POINTER, and SELECTOR, and additionally, for PL/M-386, OFFSET, HWORD, CHARINT, and SHORTINT), the context of the entire expression is always a signed integer value. Table 9-1 gives the value and type conversions for PL/M-86 and PL/M-286. Table 9-2 gives the value and type conversions for PL/M-386.

PL/M-86/286

Table 9-1 Value and Type Conversions for PL/M-86 and PL/M-286

Procedure Name	Parameter Type	Function	Result Returned
LOW	BYTE	Converts WORD value to BYTE value	BYTE value unchanged
	WORD	Converts WORD value to BYTE value	Low-order BYTE of WORD
	DWORD	Converts DWORD value to WORD value	Low-order WORD of DWORD
HIGH	BYTE		0 (zero)
	WORD	Converts WORD value to BYTE value	High-order BYTE of WORD
	DWORD	Converts DWORD value	High-order WORD of DWORD
DOUBLE	BYTE	Converts BYTE value to WORD value	WORD value, by appending 8 high-order zero bits
	WORD	Converts WORD value to DWORD value	DWORD value, by appending 16 high-order zero bits
	DWORD		DWORD value unchanged
FLOAT	INTEGER	Converts INTEGER value to REAL value	Same value of type REAL
FIX	REAL	Converts REAL value to INTEGER value (rounds toward zero)	Same value of type INTEGER if within range −32768 to +32767; otherwise undefined
INT	BYTE WORD	Converts unsigned value to INTEGER value, interprets parameter as positive	Same value of type INTEGER if within range −32768 to 32767, otherwise undefined

Table 9-1 Value and Type Conversions for PL/M-86 and PL/M-286 (continued)

Procedure Name	Parameter Type	Function	Result Returned
SIGNED	BYTE WORD	Converts an unsigned value to an INTEGER value	BYTE value is extended with 8 high-order zeros WORD value unchanged
UNSIGN	INTEGER	Converts an INTEGER value to a WORD value	Signed INTEGER value is interpreted as unsigned WORD value
ABS	REAL	Converts negative real value to positive real value	Absolute value of parameter: value unchanged if positive, – (value) if negative. Result type same as parameter type
IABS	INTEGER	Converts negative integer to positive integer	Absolute value of parameter: value unchanged if positive, – (value) if negative. Result type same as parameter type
BYTE	any unsigned type	Converts any unsigned type to BYTE	BYTE value, by truncation
	INTEGER	Converts INTEGER type to BYTE	BYTE value, by truncation
	REAL	Converts any REAL type to BYTE	BYTE (INTEGER (real))
	SELECTOR	Converts SELECTOR to BYTE	BYTE value, by truncation
	POINTER	Converts offset portion of POINTER to BYTE	BYTE (OFFSET$OF (pointer))
WORD	any unsigned type	Converts any unsigned type to WORD	WORD value, by truncation or zero extension
	INTEGER	Converts INTEGER type to WORD	WORD value, by sign extension
	REAL	Converts any real type to WORD	WORD (INTEGER (real))

Table 9-1 Value and Type Conversions for PL/M-86 and PL/M-286 (continued)

Procedure Name	Parameter Type	Function	Result Returned
WORD (continued)	SELECTOR	Converts SELECTOR to WORD	WORD value
	POINTER	Converts offset portion of POINTER to WORD	WORD (OFFSET$OF (pointer))
DWORD	any unsigned type	Converts any unsigned type to DWORD	DWORD value, by zero extension
	INTEGER	Converts INTEGER type to DWORD	DWORD value, by sign extension
	REAL	Converts any real type to DWORD	DWORD (INTEGER (real))
	SELECTOR	Converts SELECTOR to DWORD	DWORD value, by zero extension
	POINTER	Converts offset portion of POINTER to DWORD	DWORD (OFFSET$OF (pointer))
INTEGER	any unsigned type	Converts any unsigned type to INTEGER	INTEGER value, by zero extension or truncation
	INTEGER	INTEGER value not changed	INTEGER value
	REAL	Converts any real type to INTEGER	FIX (real)
	SELECTOR	Converts SELECTOR to INTEGER	INTEGER value
	POINTER	Converts offset portion of POINTER to INTEGER	INTEGER (OFFSET$OF (pointer))
REAL	any unsigned type	Converts any unsigned type to REAL	REAL (SIGNED (unsigned))
	INTEGER	Converts INTEGER type to REAL	FLOAT (signed)
	REAL		value unchanged

PL/M-86/286
(continued)

Table 9-1 Value and Type Conversions for PL/M-86 and PL/M-286 (continued)

Procedure Name	Parameter Type	Function	Result Returned
SELECTOR	any unsigned binary type	Converts any unsigned binary type to SELECTOR	SELECTOR value, by zero extension or truncation
	any signed integer data type	Converts any signed integer data type to SELECTOR	SELECTOR value by sign extension or truncation.
	POINTER		Selector portion of the POINTER
	REAL		Cannot be used.
POINTER	any unsigned type	Converts any unsigned type to POINTER	BUILD$PTR (DS, OFFSET (unsigned)) (DS is selector of current data segment)
	INTEGER	Converts INTEGER type to POINTER	BUILD$PTR (DS, OFFSET (signed)) (DS is selector of current data segment)
	SELECTOR		BUILD$PTR (SELECTOR, 0)

Notes: Conversions from REAL to OFFSET, SELECTOR, or POINTER, and vice versa, are not allowed.

end
PL/M-86/286

Table 9-2 Value and Type Conversions for PL/M-386

Procedure Name	Parameter Type	Function	Result Returned
LOW	BYTE		BYTE value unchanged
	HWORD	Converts HWORD value to BYTE value	Low-order BYTE of HWORD
	WORD or OFFSET	Converts WORD or OFFSET value to HWORD value	Low-order HWORD of WORD or OFFSET
	DWORD	Converts DWORD value to WORD value	Low-order WORD of DWORD
HIGH	BYTE		zero
	HWORD	Converts HWORD value to BYTE value	High-order BYTE of HWORD
	WORD or OFFSET	Converts WORD or OFFSET value to HWORD value	High-order HWORD of WORD or OFFSET
	DWORD	Converts DWORD value to WORD value	High-order WORD of DWORD
DOUBLE	BYTE	Converts BYTE value to HWORD value	HWORD, by appending 8 high-order zero bits
	HWORD	Converts HWORD value to WORD value	WORD, by appending 16 high-order zero bits
	WORD or OFFSET	Converts WORD or OFFSET value to DWORD value	DWORD, by appending 32 high-order zero bits
	DWORD		DWORD value unchanged
FLOAT	CHARINT SHORTINT INTEGER	Converts signed integer value to REAL value	Same value of type REAL
FIX	REAL	Converts REAL value to INTEGER value	Same value of type INTEGER if within range $-2^{**}31$ to $+(2^{**}31)-1$ otherwise undefined

Table 9-2 Value and Type Conversions for PL/M-386 (continued)

Procedure Name	Parameter Type	Function	Result Returned
INT	BYTE HWORD WORD	Converts unsigned binary value to INTEGER value, interprets parameter as positive	Same value of type INTEGER if within range −2**31 to +(2**31) − 1 otherwise undefined
SIGNED	BYTE	Converts unsigned integer value to INTEGER value	BYTE value is extended with 24 high-order zeros
	HWORD		HWORD value is extended with 16 high-order zeros
	WORD		WORD value unchanged
UNSIGN	CHARINT SHORTINT INTEGER	Converts INTEGER value to WORD value	Signed INTEGER value is interpreted as unsigned WORD value
ABS	REAL	Converts negative real value to positive real value	Absolute value of parameter: value unchanged if positive −(value) if negative. Result type is same as parameter type.
IABS	CHARINT SHORTINT INTEGER	Converts negative integer to positive integer	Absolute value of parameter: value unchanged if positive −(value) if negative. If −(value) is out of range, result is undefined. Result type is same as parameter type.
BYTE	any unsigned type	Converts any unsigned type to BYTE	BYTE value, by truncation
	any signed type	Converts any signed type to BYTE	BYTE value, by truncation
	REAL	Converts any REAL type to BYTE	BYTE (CHARINT (real))
	SELECTOR	Converts SELECTOR to BYTE	BYTE value, by truncation
	POINTER	Converts offset portion of POINTER to BYTE	BYTE (OFFSET$OF (pointer))

Table 9-2 Value and Type Conversions for PL/M-386 (continued)

Procedure Name	Parameter Type	Function	Result Returned
HWORD	any unsigned type	Converts any unsigned type to HWORD	HWORD value, by truncation or zero extension
	any signed type	Converts any signed type to HWORD	HWORD value, by truncation or sign extension
	REAL	Converts any real type to HWORD	HWORD (SHORTINT (real))
	SELECTOR	Converts SELECTOR to HWORD	HWORD type, value unchanged
	POINTER	Converts offset portion of POINTER to HWORD	HWORD (OFFSET$OF (pointer))
WORD	any unsigned type	Converts any unsigned type to WORD	WORD value, by truncation or zero extension
	any signed type	Converts any signed type to WORD	WORD value, by sign extension
	REAL	Converts any real type to WORD	WORD (INTEGER (real))
	SELECTOR	Converts SELECTOR to WORD	WORD value, by zero extension
	POINTER	Converts offset portion of POINTER to WORD	WORD (OFFSET$OF (pointer))
DWORD	any unsigned type	Converts any unsigned type to DWORD	DWORD value, by zero extension
	any signed type	Converts any signed type to DWORD	DWORD value, by sign extension
	REAL	Converts any real type to DWORD	DWORD (INTEGER (real))
	SELECTOR	Converts SELECTOR to DWORD	DWORD value, by zero extension
	POINTER	Converts offset portion of POINTER to DWORD	DWORD (OFFSET$OF (pointer))

Table 9-2 Value and Type Conversions for PL/M-386 (continued)

Procedure Name	Parameter Type	Function	Result Returned
CHARINT	any unsigned type	Converts any unsigned type to CHARINT	CHARINT value, by truncation
	any signed type	Converts any signed type to CHARINT	CHARINT value, by sign-extension
	REAL	Converts any real type to CHARINT	CHARINT (FIX(real))
	SELECTOR	Converts SELECTOR to CHARINT	CHARINT value, by truncation
	POINTER	Converts offset portion of POINTER to CHARINT	CHARINT (OFFSET$OF (pointer))
SHORTINT	any unsigned type	Converts any unsigned type to SHORTINT	SHORTINT value, by zero extension or truncation
	any signed type	Converts any signed type to SHORTINT	SHORTINT value, by sign extension
	REAL	Converts any real type to SHORTINT	SHORTINT (FIX (real))
	SELECTOR	Converts SELECTOR to SHORTINT	SHORTINT value
	POINTER	Converts offset portion of POINTER to SHORTINT	SHORTINT (OFFSET$OF (pointer))
INTEGER	any unsigned type	Converts any unsigned type to INTEGER	INTEGER value, by zero extension or truncation
	any signed type	Converts any signed type to INTEGER	INTEGER value, by sign extension
	REAL	Converts any real type to INTEGER	INTEGER (FIX (real))
	SELECTOR	Converts SELECTOR to INTEGER	INTEGER value, by zero extension
	POINTER	Converts offset portion of POINTER to INTEGER	INTEGER (OFFSET$OF (pointer))

Table 9-2 Value and Type Conversions for PL/M-386 (continued)

Procedure Name	Parameter Type	Function	Result Returned
REAL	any unsigned type (except OFFSET)	Converts any unsigned type to REAL	REAL (SIGNED (unsigned))
	any signed type	Converts any signed type to REAL	FLOAT (signed)
	REAL		value unchanged
SELECTOR	any un-signed binary type	Converts any unsigned binary type to SELECTOR	SELECTOR value, by zero extension or truncation
	OFFSET		Current data segment selector
	any un-signed integer data type	Converts any signed inte-ger data type to SELECTOR	SELECTOR value by sign extension or truncation.
	POINTER		Selector portion of the POINTER
	REAL		Cannot be used.
OFFSET	any unsigned type	Converts any unsigned type to OFFSET	OFFSET, by zero extension or truncation
	any signed type	Converts any signed type to OFFSET	OFFSET, by sign extension
	SELECTOR		zero (0)
	POINTER		OFFSET$OF (pointer)

PL/M-386
(continued)

Table 9-2 Value and Type Conversions for PL/M-386 (continued)

Procedure Name	Parameter Type	Function	Result Returned
POINTER	any unsigned type	Converts value of any unsigned type to POINTER	BUILD$PTR (DS, OFFSET (unsigned)) (DS is selector of current data segment)
	any signed type	Converts value of any signed type to POINTER	BUILD$PTR (DS, OFFSET (signed)) (DS is selector of current data segment)
	SELECTOR		BUILD$PTR (SELECTOR, 0)
	OFFSET		BUILD$PTR (DS, OFFSET) (DS is selector of current data segment)

Notes: Conversions from REAL to OFFSET, or POINTER, and vice versa, are not allowed. Under WORD32 (the default), LONGINT is equivalent to INTEGER. ADDRESS is equivalent to OFFSET.

end
PL/M-386

PL/M-86/286

9.2.1 The PL/M-86/286 LOW, HIGH, and DOUBLE Functions

In PL/M-86 and PL/M-286, the LOW built-in function converts WORD values to BYTE values and DWORD values to WORD values. LOW is activated using the following form:

LOW (*expression*)

Where:

expression	has an unsigned binary number data type value.

If *expression* has a DWORD value, LOW returns the value of the low-order (least significant) WORD of the *expression* value. If *expression* has a WORD value, LOW returns the value of the low-order (least significant) BYTE of the *expression* value. If *expression* has a BYTE value, then LOW will return this value unchanged.

The PL/M-86/286 HIGH built-in function converts WORD values to BYTE values and DWORD values to WORD values. HIGH is activated using the following form:

```
HIGH (expression)
```

Where:

expression	has an unsigned binary number data type value.

If *expression* has a DWORD value, HIGH returns the value of the high-order (most significant) WORD of the *expression* value. If *expression* has a WORD value, HIGH returns the value of the high-order (most significant) BYTE of the *expression* value. If *expression* has a BYTE value, then HIGH returns a zero.

The PL/M-86/286 DOUBLE built-in function converts BYTE values to WORD values and WORD values to DWORD values. DOUBLE is activated using the following form:

```
DOUBLE (expression)
```

Where:

expression	has an unsigned binary number data type value.

If *expression* has a BYTE value, the DOUBLE function appends 8 high-order zero bits to convert the *expression* to a WORD value and returns this WORD value. If *expression* has a WORD value, the DOUBLE function appends 16 high-order zero bits to convert the *expression* to a DWORD value and returns this DWORD value. If *expression* has a DWORD value, the DOUBLE function returns this DWORD value unchanged.

end
PL/M-86/286

9.2.2 The PL/M-386 LOW, HIGH, and DOUBLE Functions

The PL/M-386 LOW built-in function converts DWORD values to WORD values, WORD or OFFSET values to HWORD values, and HWORD values to BYTE values. LOW is activated using the following form:

```
LOW (expression)
```

Where:

expression	has an unsigned binary number data type value.

If *expression* has a DWORD value, LOW returns the value of the low-order (least significant) WORD of the *expression* value. If *expression* has a WORD or OFFSET value, LOW returns the value of the low-order (least significant) HWORD of the *expression* value. If *expression* has an HWORD value, LOW returns the value of the low-order (least significant) BYTE of the *expression* value. If *expression* has a BYTE value, LOW returns this value unchanged.

The PL/M-386 HIGH built-in function converts DWORD values to WORD values, WORD or OFFSET values to HWORD values, and HWORD values to a BYTE values. HIGH is activated using the following form:

```
HIGH (expression)
```

Where:

expression	has an unsigned binary number data type value.

If *expression* has a DWORD value, HIGH returns the value of the high-order (most significant) WORD of the *expression* value. If *expression* has a WORD or OFFSET value, HIGH returns the value of the high-order (most significant) HWORD of the *expression* value. If *expression* has an HWORD value, HIGH returns the value of the high-order (most significant) BYTE of the *expression* value. If *expression* has a BYTE value, then HIGH will return a zero.

The PL/M-386 DOUBLE built-in function converts BYTE values to HWORD values, HWORD values to WORD values, and WORD or OFFSET values to DWORD values. DOUBLE is activated using the following form:

DOUBLE (*expression*)

Where:

expression	has an unsigned binary number data type value.

If *expression* has a BYTE value, the DOUBLE function appends 8 high-order zero bits to convert the *expression* to an HWORD value and returns this HWORD value. If *expression* has an HWORD value, the DOUBLE function appends 16 high-order zero bits to convert the *expression* to a WORD value and returns this WORD value. If *expression* has a WORD or OFFSET value, the DOUBLE function appends 32 high-order bits to convert it to a DWORD value and returns this DWORD value. If *expression* has a DWORD value, the DOUBLE function returns this DWORD value unchanged.

end
PL/M-386

9.2.3 The FLOAT Function

FLOAT is a built-in REAL function that converts a signed integer data type to a real number data type. It is activated by a function reference with the following form:

FLOAT (*expression*)

Where:

expression	is a signed integer data type.

FLOAT converts the signed integer data type to the corresponding real number data type and returns the real number data type. FLOAT can be replaced with REAL (*expression*).

9.2.4 The FIX Function

FIX is a built-in INTEGER function that converts a REAL value to an INTEGER value. It is activated by a function reference with the following form:

FIX (*expression*)

Where:

expression	has a REAL value.

FIX rounds the REAL value to the nearest INTEGER. If both INTEGER values are equally near, FIX rounds to the even value. The resulting INTEGER value is then returned.

For example:

```
FIX(1.4)      /* would result in the INTEGER value 1, */
FIX(-1.8)                                 /* in -2, */
FIX(3.5)                              /* in 4, and */
FIX(6.5)                                   /* in 6. */
```

If the result calculated by FIX is not within the implemented range of INTEGER values, the result is undefined.

NOTE

FIX is affected by the rounding mode, see Section 10.6. The default mode (round to the nearest or even value) is used in the previous examples.

FIX can be replaced with INTEGER (*expression*).

9.2.5 The INT Function

INT is a built-in INTEGER function that converts an unsigned binary data type, excluding DWORD, to a signed integer data type. It is activated by a function reference with the following form:

INT (*expression*)

Where:

expression	has an unsigned binary data type, excluding DWORD.

INT interprets the *expression* value as a positive number and returns the corresponding INTEGER value.

If the result calculated by INT is not within the implemented range of INTEGER values, the result is undefined. (See Chapter 5 for ranges for INTEGER values.)

9.2.6 The SIGNED Function

For PL/M-86 and PL/M-286, SIGNED is a built-in INTEGER function that converts a BYTE or WORD value to an INTEGER value. For PL/M-386, SIGNED is a built-in INTEGER function that converts a BYTE, HWORD, or WORD value to an INTEGER value. SIGNED is activated by a function reference with the following form:

SIGNED (*expression*)

Where:

expression	is an unsigned binary number data type, excluding DWORD.

For PL/M-86 and PL/M-286, if the expression has a BYTE value, it will be extended by 8 high-order 0 bits to produce a WORD value.

For PL/M-386, if *expression* has a BYTE or HWORD value, it will be extended by 24 or 16 high-order 0 bits, respectively, to produce a WORD value.

SIGNED interprets the WORD value as a 16-bit two's-complement number (for PL/M-86 and PL/M-286) or a 32-bit two's-complement number (for PL/M-386) and returns the corresponding integer value.

If the highest-order (most significant) bit of the WORD value is a 0, SIGNED interprets the WORD value as a positive number and returns the corresponding INTEGER value. For example:

```
SIGNED (0000$0000$0000$0100B)
```

returns an INTEGER value of 4.

If the highest-order bit of the WORD value is a 1, SIGNED returns a negative INTEGER value whose absolute magnitude is the two's complement of the WORD value. For example:

```
SIGNED(1111$1111$1111$1100B)
```

returns an INTEGER value of −4.

SIGNED can be replaced by INTEGER (*expression*).

9.2.7 The UNSIGN Function

The UNSIGN built-in function converts a signed integer data type to a WORD value. It is activated by a function reference with the following form:

UNSIGN (*expression*)

Where:

expression	is a signed integer data type.

UNSIGN converts the INTEGER value to a WORD value.

If the INTEGER value is positive, the WORD value will be numerically the same as the INTEGER value. However, if the INTEGER value is negative, the WORD value will be the two's complement of the absolute magnitude of the INTEGER value. For example:

UNSIGN(−4)

returns a WORD value of:

1111$1111$1111$1100B

UNSIGN can be replaced by WORD (*expression*).

9.2.8 The Unsigned Binary Data Type Built-in Functions

The unsigned binary data type built-in functions convert any expression to the specified unsigned binary data type. For example, the WORD and DWORD built-in functions convert any expression to a WORD or DWORD value, respectively.

The built-in functions are activated with the form:

built-in (*expression*)

Where:

built-in	is the name of the data type to which the given expression is converted (e.g., BYTE or WORD).
expression	has any value.

For example, WORD (INT1) converts the value of INT1 to a WORD value.

If *expression* is an unsigned binary number data type, it is converted by truncation or zero extension, if necessary. If *expression* is a signed integer data type, it is converted by truncation or zero extension, if necessary. If *expression* is a selector data type, it is converted by truncation or zero extension. If *expression* is a pointer data type, the offset portion of the pointer is converted by truncation or zero extension; the selector portion of the pointer is discarded. If *expression* is a real number data type, it is first converted to a signed integer using the numeric coprocessor's real to integer conversion, then the resulting value is converted to the unsigned binary number data type by truncation, if necessary.

9.2.9 Signed Integer Data Type Built-in Function

The signed integer data type built-in function converts any expression to a signed integer data type.

For example:

```
INTEGER (D)
```

converts the value of D to an INTEGER value within the INTEGER range.

If *expression* is an unsigned binary number or selector data type, it is converted by truncation or zero extension. If *expression* is a pointer data type, the offset portion of the pointer is converted by truncation or zero extension; the selector portion of the pointer is discarded. If *expression* is a real number data type, it is converted using the numeric coprocessor's real to integer conversion. For PL/M-86 and PL/M-286, if *expression* is a signed integer data type, no operation is necessary.

PL/M-386

Specific to PL/M-386, if *expression* is a signed type, it is converted by sign extension. Shorter data types are converted into longer data types by sign extending the shorter data type value. Longer data types are converted into shorter data types by sign extension of the bits equivalent to the shorter data type. For example, if a CHARINT built-in is used to convert an INTEGER value, the least significant 8 bits are sign extended and the value returned is guaranteed to be in the CHARINT range.

end PL/M-386

9.2.10 REAL Built-in Functions

The REAL built-in function converts an expression to a REAL value. Expressions of type SELECTOR, OFFSET, and POINTER cannot be used. The conversion is done using the numeric coprocessor's INTEGER to REAL conversion. If the expression is an unsigned binary number data type it is zero extended, if necessary, and interpreted as a signed value.

9.2.11 The SELECTOR Built-in Function

The SELECTOR built-in function converts any expression (except the real number data type) to a SELECTOR value. If *expression* is any unsigned binary number data type, except OFFSET, it is truncated or zero extended to 16 bits. If *expression* is a signed integer data type, it is truncated or sign extended to 16 bits. If *expression* is of type POINTER, the selector portion of the pointer is returned. If *expression* is of type OFFSET, the current data segment selector is returned. Expressions of type REAL cannot be used.

9.2.12 The POINTER Built-in Function

The POINTER built-in function converts any expression (except the real number data type) to a POINTER value. If *expression* is any unsigned binary number or signed integer data type, it is converted to type OFFSET by truncation, zero, or sign extension, if necessary. This OFFSET value is combined with the SELECTOR value of the current data segment to create the POINTER value. If *expression* is of type SELECTOR, it is combined with an OFFSET value of zero to create the POINTER value. Expressions of type REAL cannot be used.

9.2.13 The OFFSET Built-in Function

The OFFSET built-in function converts any expression (except the real number data type) to an OFFSET value. If *expression* is any unsigned binary number or signed integer data type, it is converted to type OFFSET by truncation, or by zero or sign extension. If *expression* is of type SELECTOR, an OFFSET value of zero is returned. If the *expression* is of type POINTER, the offset portion of the pointer is returned. ADDRESS values are equivalent to OFFSET. Expressions of type REAL cannot be used.

9.2.14 The ABS and IABS Functions

The ABS built-in function returns the absolute value of a real number data type. It is activated by a function reference with the following form:

ABS (*expression*)

Where:

expression	is a real number data type.

If the value of *expression* is positive, ABS returns it unchanged. If the value of *expression* is negative, ABS returns −(*expression*).

The IABS built-in function returns the absolute value of a signed integer data type. It is activated by a function reference with the following form:

IABS (*expression*)

Where:

expression	is a signed integer data type.

If the value of *expression* is positive, IABS returns it unchanged. If the value of *expression* is negative, IABS returns −(*expression*).

9.3 Shift and Rotate Functions

With the shift and rotate functions, bit patterns can be moved to the right and to the left. In a shift, bits moved off one end of the pattern are lost, and zero bits move into the pattern from the other end (except in the case of the algebraic shift right function, SAR; see Section 9.3.3). In a rotate, bits moved off one end of the pattern are moved onto the other end of the pattern. It is not possible to perform a rotate on a signed integer algebraic pattern.

For PL/M-86 and PL/M-286, in shift and rotate operations, a value is handled as a pattern of 8 bits (for a BYTE value), 16 bits (for a WORD or INTEGER value), or 32 bits (for a DWORD value). For PL/M-386, a value is handled as a pattern of 8 bits (for a BYTE or CHARINT value), 16 bits (for a HWORD or SHORTINT value), 32 bits (for WORD, OFFSET, or INTEGER values), or 64 bits (for a DWORD value). The pattern is moved to the right or left by a specified number of bits called the bit count.

9.3.1 Rotation Functions

The type of the rotate left (ROL) and rotate right (ROR) built-in functions depends on the type of expression given as an actual parameter. These built-ins are activated by function references with the following forms:

ROL (*pattern, count*)
ROR (*pattern, count*)

Where:

pattern and *count*	are expressions using an unsigned binary number data type.

If *count* is any unsigned binary number data type except BYTE, all but the low-order bits will be dropped to produce a BYTE value. If the value of *count* is 0, no rotation occurs.

The value of *pattern* is handled as an 8-bit, 16-bit, 32-bit, or 64-bit quantity. The type of *pattern* determines which of the unsigned binary number data types is used. This, in turn, determines the value of *pattern*. For example, HQWORD, SHORTINT, and WORD are all 16-bit quantities (see Table 3-2). The number of bit positions by which *pattern* is rotated is specified by *count*.

The following are examples of the action of these procedures:

```
ROR(10011101B,1)                 returns a value of 11001110B
ROL(10011101B,2)                 returns a value of 01110110B
ROR(1101011010011010B,9)         returns a value of 0100110101101011B
```

9.3.2 Logical-Shift Functions

The type of the logical-shift left (SHL) and logical-shift right (SHR) built-in functions depends on the type of the expression given as an actual parameter. SHL and SHR are activated by function references with the forms:

SHL (*pattern, count*)
SHR (*pattern, count*)

Where:

pattern and *count*	are expressions using an unsigned binary number data type.

If *count* is any unsigned binary number data type except BYTE, all but the 8 low-order bits will be dropped to produce a BYTE value. If the value of *count* is 0, no shift occurs.

The value of *pattern* can be a BYTE, HWORD, WORD, or DWORD value and the value will not be converted. If *pattern* is a BYTE value, the function will return a BYTE value. If *pattern* is an HWORD value, the function will return an HWORD value. If *pattern* is a WORD value, the function will return a WORD value; if *pattern* is a DWORD value, the function will return a DWORD value.

The value of pattern is shifted left (by SHL) or right (by SHR), with the bit count given by *count*.

A shift operation can force one bit out of the pattern. For example:

```
SHL(1000$0001B,1)
```

returns 0000$0010B, losing the former high-order bit, and:

```
SHR(1000$0001B,1)
```

becomes 0100$0000B, losing the former low-order bit.

If the specified *pattern* and *count* do not lose information, a shift of one bit position has the effect of multiplication by two for a left shift, or division by two for a right shift. For example, suppose that VAR is a BYTE variable with a value of eight. This is represented as 0000$1000B. SHL(VAR,1) would return 0001$0000B, which represents 16, and SHR(VAR,1) would return 0000$0100B, which represents four.

Casting can be used to ensure that no information is lost in a shift, as in the following example:

```
SHL(WORD(LIT$MASK),3)
```

9.3.3 Algebraic-Shift Functions

The type of the algebraic-shift left (SAL) and algebraic-shift right (SAR) built-in functions depends on the type of the expression given as an actual parameter. SAL and SAR are activated by function references with the following forms:

SAL (*pattern, count*)
SAR(*pattern, count*)

Where:

pattern	is an expression using a signed integer data type.
count	is an expression using an unsigned binary data type.

If *count* is any unsigned binary data type except BYTE, all but the 8 low-order bits will be dropped to produce a BYTE value. If the value of *count* is zero, no shift occurs.

For PL/M-386, the type of *pattern* can be a CHARINT, SHORTINT, or INTEGER value. All values are converted to INTEGER before the shift operations, and an INTEGER value is returned.

For PL/M-86 and PL/M-286, SAL and SAR treat the INTEGER value of *pattern* as a bit pattern. For PL/M-386, SAL and SAR treat the signed value of *pattern* as a bit pattern. This pattern is shifted to the left or to the right.

In a left shift (SAL), zero-bits move into the pattern from the right (as in SHL and SHR).

In a right shift (SAR), either zero bits or one bits move into the pattern from the left. If the original value of pattern is positive, the sign bit (leftmost bit) is a 0, and zero bits move in from the left. If the original value is negative, the sign bit is a 1, and one bits move in from the left.

In some instances (as in logical shifts), an algebraic shift of one bit position can have the effect of multiplication by two for a left shift or division by two for a right shift. For example, suppose that VAL is an INTEGER variable with a value of −8. This value is 1111$1111$1111$1000B. SAL(VAL,1) would return 1111$1111$1111$0000B, which is −16, and SAR(VAL,1) would return 1111$1111$1111$1100B, which is −4.

PL/M-386

9.3.4 Concatenate Functions

The concatenate functions (SHLD and SHRD) are built-in WORD double-shift functions that concatenate two WORD values to form a 64-bit string, shift the concatenated pattern left (SHLD) or right (SHRD) by *count* bits, and return the destination WORD. These built-ins are activated by function references with the following form:

keyword (*high pattern, low pattern, count*)

Where:

keyword	is SHLD or SHRD.
high pattern	is a WORD value.
low pattern	is a WORD value.
count	is a BYTE, HWORD, or WORD value that determines how many bits to shift the concatenated pattern.

SHLD concatenates the bit pattern of the WORD value *high pattern* with the bit pattern of the WORD value *low pattern* to form a 64-bit string. *high pattern* is placed in the high 32 bits and *low pattern* is placed in the low 32 bits. The concatenated pattern is shifted left by the number of bits given by *count* MODULO 32. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. This has the effect of shifting the high order bits of *low pattern* into the low order bits of *high pattern*. SHLD returns the high 32 bits of the shifted pattern.

SHRD concatenates the bit pattern of the WORD value *high pattern* with the bit pattern of the WORD value *low pattern* to form a 64-bit string. *high pattern* is placed in the high 32 bits and *low pattern* is placed in the low 32 bits. The concatenated pattern is shifted right by the number of bits given by count MODULO 32. These operands are taken MODULO 32 to provide a number between 0 and 31 by which to shift. This has the effect of shifting the low order bits of *high pattern* into the high order bits of *low pattern*. SHRD returns the low 32 bits of the shifted pattern.

9.4 String Manipulation Procedures and Functions

The term string is used here in a broader sense than previously, in which string was used to refer to a BYTE string. In this section, a string is any contiguously stored set of unsigned binary number data type values (excluding DWORD and OFFSET). A string can be regarded as if it were an unsigned binary number type (excluding DWORD and OFFSET) array, and the array items can be referred to as elements.

The word index refers to the position of a given element within a string. The index is similar to the subscript of an array reference. Thus, the index of the first element of a string is 0, the index of the second element is 1, and so on.

In the following descriptions, the location of a string always means the location of its first element. In each string manipulation procedure, the location of a string is specified by a parameter called *source* or *destination*, which is an expression with a POINTER value. The *source* points to the lowest element. For example, with MOVB and MOVW, the lowest element (element 0) is the first element to be processed. With MOVRB and MOVRW, the lowest element is the last element to be processed, as discussed in the following sections.

The length of a string is the number of elements it contains. In each string manipulation procedure, the number of elements to be processed is specified by a parameter called *count*.

NOTE

If the *source* or *destination* string address is in SELECTOR or WORD form, use the @ operator of a variable based on the address. Otherwise, the built-in function BUILD$PTR can be used to construct the pointer-parameter for the string built-in.

For PL/M-86 and PL/M-286, each of the string manipulation procedures described in the following sections (except XLAT) is available in pairs, that is, each built-in is available for BYTE and WORD strings.

For PL/M-386, each of the string-manipulation procedures described in the following sections (except XLAT) is available in triples, that is, each built-in is available for BYTE, HWORD, and WORD strings.

9.4.1 The Copy String in Ascending Order Procedure

MOVxx is an untyped procedure that copies a string of length *count* from one location to another. It is activated by a CALL statement with the following form:

CALL *keyword* (*source, destination, count*);

Where:

	PL/M-86	**PL/M-286**	**PL/M-386**
keyword	MOVB, MOVW	MOVB, MOVW	MOVB, MOVHW, MOVW
source and *destination*	expressions with POINTER values	expressions with POINTER values	expressions with POINTER values
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

The string elements are copied in ascending order (i.e., element 0 is copied first, then element 1, etc.). This order is significant if the *source* string and the *destination* string overlap. If the value of *destination* is higher than the value of *source*, and the two strings overlap, elements in the overlap area will be overwritten before they are copied. To avoid the overwriting, use MOVRxx instead of MOVxx.

MOVW performs the same function as MOVB except that MOVW copies a WORD string instead of a BYTE string.

For PL/M-386, MOVHW performs the same function as MOVB except that MOVHW copies an HWORD string.

9.4.2 The Copy String in Descending Order Procedure

MOVRxx is an untyped procedure that copies a string of length *count* from one location to another. It is activated by a call statement with the following form:

CALL *keyword* (*source*, *destination*, *count*);

Where:

	PL/M-86	PL/M-286	PL/M-386
keyword	MOVRB, MOVRW	MOVRB, MOVRW	MOVRB, MOVRHW, MOVRW
source and *destination*	expressions with POINTER values	expressions with POINTER values	expressions with POINTER values
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value.

The MOVRB built-in procedure is similar to the MOVB procedure except that the elements in the MOVRB source string are copied to the *destination* string in descending order (i.e., element (count-1) is copied first, then element (count-2), and so on, with element 0 copied last). This order is significant when the two strings overlap. If the value of *destination* is higher than the value of *source*, and an overlap exists, elements in the overlap area will not be overwritten until they have been copied. However, if the value of *source* is higher than the value of *destination*, elements in the overlap area will be overwritten before they are copied.

MOVRW performs the same function as MOVRB, except MOVRW copies a WORD string instead of a BYTE string.

For PL/M-386, MOVHW performs the same function as MOVRB except that MOVHW copies an HWORD string.

NOTE

If two strings overlap, use a procedure such as the following (written in PL/M-86) to make the correct choice between MOVB and MOVRB. This ensures that elements in the overlap area will not be overwritten until after they have been copied.

```
MOVBYTES: PROCEDURE (SRC, DST, CNT);
  DECLARE (SRC, DST) POINTER, CNT WORD;
  IF SRC>DST THEN CALL MOVB (SRC, DST, CNT);
  ELSE CALL MOVRB (SRC, DST, CNT);
END MOVBYTES;
```

This procedure can be activated without the need to consider whether overlap may occur or whether *source* or *destination* is higher.

9.4.3 The Compare String Function

CMPxx is a built-in WORD function that compares two strings of length *count*. It is activated by a function reference with the following form:

keyword (*source1, source2, count*)

Where:

	PL/M-86	**PL/M-286**	**PL/M-386**
keyword	CMPB, CMPW	CMPB, CMPW	CMPB, CMPHW, CMPW
source1 and *source2*	expressions with POINTER values	expressions with POINTER values	expressions with POINTER values
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

CMPB compares two BYTE strings of length *count* whose locations are *source1* and *source2*.

If every element in the string at *source1* is equal to the corresponding element in the string at *source2*, CMPxx returns a WORD value (0FFFFH for PL/M-86 and PL/M-286, and 0FFFFFFFFH for PL/M-386. Otherwise, CMPxx returns the index (position within the strings) of the first pair of elements found to be unequal.

CMPW performs the same as function CMPB except that CMPW compares two WORD strings instead of two BYTE strings.

For PL/M-386, CMPHW performs the same function as CMPB, except that CMPHW compares two HWORD strings.

9.4.4 The Find Element Functions

FIND is a built-in WORD function that searches a string to find an element that has a specified value. It is activated by a function reference of the following form:

keyword (source, target, count)

Where:

	PL/M-86	PL/M-286	PL/M-386
keyword	FINDB, FINDW, FINDRB, FINDRW	FINDB, FINDW, FINDRB, FINDRW	FINDB, FINDHW, FINDW, FINDRB, FINDRHW, FINDRW
source	expression with POINTER value	expression with POINTER value	expression with POINTER value
target	expression with BYTE or WORD value — the high-order bits are dropped to produce a BYTE or WORD value	expression with BYTE or WORD value — the high-order bits are dropped to produce a BYTE or WORD value	expression with BYTE, HWORD, or WORD value — the high-order bits are dropped to produce a BYTE, HWORD, or WORD value
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

FINDB examines each element of the source string (in ascending order) until it finds an element whose value matches the BYTE value of *target*, or until *count* elements have been searched, with none of them having matched the *target*. If the search is successful, FINDB returns the index of the first element of the string that matches *target*. If the search is unsuccessful, FINDB returns a WORD value.

For PL/M-386, FINDHW performs the same function as FINDB, except that FINDHW searches an HWORD string. If *target* has a BYTE value, it is extended by

8 high-order, 0-bits to produce an HWORD value. If *target* has a WORD value, it is truncated by 16 high-order bits to produce an HWORD value.

FINDW performs the same function as FINDB, except that FINDW searches a WORD string. For PL/M-86 and PL/M-286, if *target* has a BYTE value, it is extended by 8 high-order 0-bits to produce a WORD value. For PL/M-386, if *target* has a BYTE or HWORD value, *target* is extended appropriately to produce a WORD value.

FINDRB performs the same function as FINDB, except that FINDRB searches the *source* string in descending order. Thus, if each search is successful, FINDRB returns the index of the last (highest subscript) element that matches the BYTE value of *target*. FINDRW performs the same function as FINDRB, except that FINDRW searches a WORD string (in descending order).

For PL/M-386, FINDRHW performs the same function as FINDRB, except that FINDRHW searches an HWORD string (in descending order).

9.4.5 The Find String Mismatch Function

SKIP is a built-in WORD function that searches the BYTE string of length *count* at a specified location (given by *source*) for the first BYTE value that does not match the target BYTE. This search begins with the first BYTE value of the string. The result is either a WORD value (0FFFFH for PL/M-86 and PL/M-286, and 0FFFFFFFFH for PL/M-386) if the string contains only BYTE values equal to the target BYTE, or the WORD value will be equal to the index of the first BYTE value not equal to the target BYTE.

The function is activated by a function reference of the following form:

keyword (source, target, count)

Where:

	PL/M-86	PL/M-286	PL/M-386
keyword	SKIPB, SKIPW, SKIPRB, SKIPRW	SKIPB, SKIPW, SKIPRB, SKIPRW	SKIPB, SKIPHW, SKIPW, SKIPRB, SKIPRHW, SKIPRW
source	expression with POINTER value	expression with POINTER value	expression with POINTER value
target	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, WORD, or HWORD value
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

SKIPW performs the same function as SKIPB, except that SKIPW searches a WORD source string to find the first element that does not match the WORD value of target.

SKIPRB searches a BYTE string of the length specified by *count*, at the location given by *source*, for the last BYTE value that does not match the target BYTE. This search begins with the last BYTE value in the string. The result is a WORD value (0FFFH for PL/M-86 and PL/M-286, and 0FFFFFFFFH for PL/M-386) if the string contains only BYTE values equal to the target BYTE, or the index of the last BYTE value, if the last BYTE value in the string is not equal to the target BYTE.

SKIPRW performs the same function as SKIPRB, except that SKIPRW searches for the last element in the WORD source string that does not match the WORD value of the target.

For PL/M-386, SKIPHW performs the same function as SKIPB, except that SKIPHW searches an HWORD source string to find the first element that does not match the HWORD value of target. SKIPRHW also performs the same function as SKIPRB, except that SKIPRHW searches for the last element in the HWORD source string that does not match the HWORD value of the target.

9.4.6 The Translate String Procedure

XLAT is an untyped procedure that uses a translation table to translate a BYTE string to produce another BYTE string. It is activated by a CALL statement of the form:

```
CALL XLAT (source, destination, count, table)
```

Where:

	PL/M-86	**PL/M-286**	**PL/M-386**
source, *destination*, and *table*	expressions with POINTER values	expressions with POINTER values	expressions with POINTER values
count	expression with WORD value	expression with WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

XLAT translates the *count* BYTE elements in the *source* string, placing the translated elements in the *destination* string. The value of *table* is assumed to be the location of a BYTE string of up to 256 elements. This string is used as a translation table.

The value of an element in the *source* string is used as an index into the translation table. The index selects one element from the translation table; this element is then copied into the destination string.

For example, if the fifth element in the source string is 202, then 202 is used as an index for the translation table. The 203rd element of the table is copied into the fifth position in the *destination* string.

The elements of the *source* string are translated into the *destination* string in ascending order.

9.4.7 The Set String to Value Procedure

The SET built-in is an untyped procedure that sets each element of a BYTE string, the length of which is specified by *count*, to a single specified value. SET is activated by a CALL statement with the following form:

```
CALL keyword (newvalue, destination, count)
```

Where:

	PL/M-86	PL/M-286	PL/M-386
keyword	SETB, SETW	SETB, SETW	SETB, SETHW, SETW
newvalue	expression with BYTE or WORD value — the high-order bits are dropped to produce a BYTE or WORD value	expression with BYTE or WORD value — the high-order bits are dropped to produce a BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value — the high-order bits are dropped to produce a BYTE, WORD, or HWORD value
destination	expression with POINTER value	expression with POINTER value	expression with POINTER value
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

SETB assigns the BYTE value of *newvalue* to each element of a BYTE string.

SETW performs the same function as SETB except that SETW assigns a single WORD value to each element of a WORD string. For PL/M-86 and PL/M-286, if *newvalue* has a BYTE value, it will be extended by 8 high-order zero bits to produce a WORD value. For PL/M-386, if *newvalue* has a BYTE or an HWORD value, it will be extended by 24 or 8 high-order 0 bits, respectively, to produce a WORD value.

For information on WORD32/WORD16 mapping, see Tables 9-3, 10-1, and 11-3.

9.5 PL/M-386 Bit Manipulation Built-Ins

9.5.1 The Copy BIT String Procedure

MOVBIT is an untyped built-in procedure that copies a bit string of length *count* from one location to another. In memory, bits are numbered starting from the right. The right-most bit is the most significant bit. The left-most bit is the least significant bit. MOVBIT is activated by a CALL statement with the following form:

```
CALL MOVBIT (sbase, sbitoffset, dbase, dbitoffset, count);
```

Where:

sbase and *dbase*	are expressions with POINTER values.
sbitoffset, *dbitoffset* and *count*	are expressions with BYTE, HWORD, OFFSET, or WORD values.

The MOVBIT built-in procedure moves the number of bits specified by *count* from the bit location given by the base address *sbase*, and the bit offset *sbitoffset* from that base to the location given by the base address *dbase* and the bit offset *dbitoffset*. These bits are moved beginning with the low-order bit (least significant bit).

The MOVRBIT built-in procedure performs the same function as the MOVBIT built-in, except that MOVRBIT moves bits, in descending order, beginning with the high-order bit (most significant bit).

9.5.2 The Find Set BIT Function

SCANBIT is a built-in WORD function that searches a bit string to find a set bit, (i.e., a bit with the value of 1). In memory, bits are numbered starting from the right. The right-most bit is the most significant bit. The left-most bit is the least significant bit. SCANBIT is activated by a function reference with the following form:

```
SCANBIT (sbase, sbitoffset, count)
```

Where:

sbase	is an expression with a POINTER value.
sbitoffset, *count*	are expressions with BYTE, HWORD, OFFSET, or WORD values.

PL/M-386
(continued)

The SCANBIT built-in function searches the bit string of length *count* at the bit location given by the base address *sbase* and the bit offset *sbitoffset* from that base for the first set bit, beginning with the low-order bit (least significant bit) in the string. The result of SCANBIT is either a WORD value of 0FFFFFFFFH if the string contains all 0 bits, or the index of the first set bit.

SCANRBIT performs the same function as SCANBIT, except that SCANRBIT starts at the high-order bit (most significant bit) in the string and searches for a set bit, in descending order, and returns the location of the first set bit it encounters. The result of SCANRBIT is either a WORD value of 0FFFFFFFFH if the string contains all 0-bits, or the index of the first set bit encountered.

end
PL/M-386

9.6 Miscellaneous Built-Ins

9.6.1 The Move BYTES Procedure

MOVE is an untyped procedure that moves the number of bytes specified by *count* to the location given by the value of *destination*, starting at the location given by the value of *source*. If the *source* and *destination* fields overlap, the result is undefined. MOVE is provided for compatibility with PL/M-80 programs. MOVE is activated by a CALL statement with the following form:

CALL MOVE (*count, source, destination*)

Where:

	PL/M-86	PL/M-286	PL/M-386
count	expression with WORD or BYTE value	expression with WORD or BYTE value	expression with BYTE, HWORD, OFFSET, or WORD value
source and *destination*	expressions with WORD values	expressions with WORD values	expressions with OFFSET values

If either *source* or *destination* is a value other than the value specified in the preceding syntax description, the value will be extended by high-order 0 bits to produce the appropriate value. The values of *source* and *destination* are assumed to be the addresses of the *source* string and the *destination* string.

The operation of the MOVE procedure differs from the MOVB procedure (see Section 9.4.1), as follows:

- The *source* and *destination* parameters must have the specified value. If they are other unsigned values, they will be converted to the correct value. POINTER values cannot be used, nor can values be supplied with the @ operator. Thus, MOVE can only handle strings whose locations can be expressed as the specified *source* and *destination* value addresses.
- The parameters are in a different order than the one used by the other built-in string functions.
- The results are always undefined if the *source* and *destination* strings overlap.

9.6.2 The Time Delay Procedure

TIME is an untyped built-in procedure that causes a time delay specified by its actual parameter. TIME is activated by a CALL statement with the following form:

CALL TIME (*expression*);

where the *expression* is converted, if necessary, to a WORD quantity for PL/M-86 and PL/M-286 and to an HWORD quantity for PL/M-386. The length of time measured by the procedure is a multiple of 100 microseconds. If the actual parameter evaluates to *n*, then the delay caused by the procedure is 100*n* microseconds. For example, the statement:

```
CALL TIME (45);
```

causes a delay of 4.5 milliseconds. For PL/M-86 and PL/M-286, the maximum delay is 6.55 seconds. For PL/M-386, the maximum delay is 12 hours. If required, longer delays can be obtained by repeated activations. The following block takes one second to execute:

```
DO I = 1 TO 40;
          CALL TIME (250);
END;
```

The TIME procedure is based on the microprocessor's CPU cycle times. The TIME procedure assumes 8 MHz for the 8086 and the 80286 microprocessors and 16 MHz for the 80386 microprocessor.

PL/M-86

9.6.3 The Memory Reference Array

MEMORY is a BYTE array of unspecified length which represents an uninitialized (free) segment of the 8086 microprocessor's storage. References to MEMORY can be subscripted. The maximum subscript depends on both the system environment and the program. References to MEMORY, either subscripted or unqualified, can be used in location references. For example, @ MEMORY is the location of the beginning of free memory space (i.e., byte 0 of the memory segment).

A reference to MEMORY cannot be used as an actual parameter for the LENGTH, LAST, and SIZE procedures (see Sections 9.1.1 through 9.1.3). However, some systems provide service routines to determine the size of free memory.

end PL/M-86

9.6.4 The Lock Set Function

LOCKSET is a built-in BYTE function that enables implementation of a simple software synchronization lock. It is called by a function reference with the following form:

LOCKSET (*lockptr, newvalue*)

Where:

	PL/M-86	PL/M-286	PL/M-386
lockptr	expression with POINTER value	expression with POINTER value	expression with POINTER value
newvalue	expression with BYTE or WORD value — the high-order bits are dropped to produce a BYTE value	expression with BYTE or WORD value — the high-order bits are dropped to produce a BYTE value	expression with BYTE, HWORD, or WORD value — the high-order bits are dropped to produce a BYTE value

The action of LOCKSET is as follows: the *lockptr* parameter is used as a pointer to a BYTE variable; the value of *newvalue* is assigned to this variable, and LOCKSET returns the original value of the variable. During this transaction, the microprocessor's CPU prevents any other process from accessing the same memory location.

To see how this facility can be used, assume a system has more than one microprocessor using the same memory, and has a program in one of these microprocessors. This program uses memory locations that are also used by other microprocessors in the system.

Within certain critical regions of the program, it is critical that no other microprocessor can access the shared memory locations. To achieve this, declare a global BYTE variable called LOCK, and establish a convention that if LOCK = 0, any microprocessor in the system can access the shared memory locations. However, if LOCK = 1, no microprocessor can access the shared memory locations except for the microprocessor that set LOCK to 1.

Write the function reference LOCKSET(@LOCK,1). The value 1 will be assigned to LOCK. If the value returned by LOCKSET is 0, then LOCK has not been set, and this microprocessor is the one that set it. At the end of the critical region, the lock must be released by writing LOCK = 0.

If LOCKSET returns a value of 1, then LOCK has been set and this microprocessor was not the one that set LOCK. Wait until a LOCKSET(@LOCK,1) function reference returns a value of 0 before accessing the shared memory locations.

Thus, the program could contain the following construction:

```
                                                /*Begin critical region*/
    DO WHILE LOCKSET(@LOCK,1);
             /*Do nothing but repeat until LOCKSET returns 0*/
    END;
         /*Now LOCK has been set to 1 by this microprocessor*/
    .  .  .
                      /*Critical region of program, where shared
                               memory locations are accessed*/
    .  .  .
    LOCK=0;
                                                  /*End critical region*/
```

In the simple case just described, only one software lock is used. It is represented by the variable LOCK. But, if more than one set of memory locations needed protection at different times, it is possible to establish as many different software locks as necessary, with each lock using a different BYTE variable.

Also, note that a software lock can be used for purposes other than protecting memory locations. LOCKSET provides a mechanism that can be used to implement various types of synchronization in a multiprocessor system.

PL/M-386

9.6.5 The Lock BIT Function

Lock BIT is a built-in BYTE function that is similar to the LOCKSET built-in described in the previous section. It is a BYTE procedure called by a function reference with the form:

keyword (*bbase, boffset*)

Where:

keyword	is BITLOCKSET, BITLOCKRESET, or BITLOCKCOMPLEMENT.
bbase	is an expression with a POINTER value.
boffset	is an expression with a BYTE, HWORD, or WORD value.

The action of BITLOCKSET is as follows: the *bbase* and *boffset* parameters are used as the base address and bit offset to point to a certain bit in memory. The value 1 is assigned to this variable, and BITLOCKSET returns a BYTE. The returned value is TRUE (0FFH) if the original content of the bit was 1, otherwise it is FALSE. During this transaction, the microprocessor's CPU prevents any other process from accessing the same memory location. BITLOCKRESET performs the same function as BITLOCKSET, except that BITLOCKRESET assigns the value 0 to the bit variable. BITLOCKCOMPLEMENT performs the same function as BITLOCKSET, except that BITLOCKCOMPLEMENT complements the BYTE variable; that is, if the value was initially 0, it is set to 1 and vice versa.

end **PL/M-386**

PL/M-86

9.7 Interrupt-Related Procedures

The two capabilities described in this section enable PL/M-86 programs to set interrupt vectors and determine the entry point of an interrupt-handling procedure. For detailed information concerning interrupt handling procedures, see the 8086 reference literature.

9.7.1 The Set Interrupt Vector Procedure

The SET$INTERRUPT procedure enables an executing program to set an interrupt vector to point to the interrupt entry point of a separately compiled interrupt handling routine, or to alter such vectors dynamically. See also Section 8.5.1 and Appendix G.

The procedure is invoked by a CALL of the following form:

CALL SET$INTERRUPT (*constant,name*)

where *name* is the interrupt procedure name, and *constant* is an interrupt number (i.e., a whole-number constant between 0 and 255). *Name* must be a previously declared interrupt procedure.

PL/M-86
(continued)

9.7.2 The Return Interrupt Entry Point Function

The INTERRUPT$PTR built-in function returns the interrupt entry point. INTERRUPT$PTR has the following form:

INTERRUPT$PTR (*name*)

INTERRUPT$PTR is typically used in an assignment statement, for example:

```
INT$ARRAY(4) = INTERRUPT$PTR (HANDLER_PROC_4)
```

The interrupt entry point is not accessible without using this function, since the @ operator refers to the procedure entry point instead. These differences are discussed in greater detail in Appendix G. *Name* must be a previously declared interrupt procedure.

end
PL/M-86

9.8 POINTER and SELECTOR-Related Functions

With the following built-in functions, programs can manipulate POINTER and SELECTOR values that serve as location addresses in the microprocessor's memory.

9.8.1 The Return POINTER Value Function

BUILD$PTR is a built-in POINTER function that takes the specified *segment* and *offset* value and returns a POINTER value. It is activated by a function reference with the following form:

BUILD$PTR (*segment, offset*)

Where:

	PL/M-86	PL/M-286	PL/M-386
segment	expression with SELECTOR value	expression with SELECTOR value	expression with SELECTOR value
offset	expression with WORD value	expression with WORD value	expression with OFFSET value

9.8.2 The Return Segment Portion of POINTER Function

SELECTOR$OF is a built-in SELECTOR function that returns the segment portion of a POINTER. It is activated by a function reference with the following form:

```
SELECTOR$OF (pointer)
```

Where:

pointer	is an expression with a POINTER value.

9.8.3 The Return Offset Portion of POINTER Function

OFFSET$OF returns the offset portion of a POINTER. For PL/M-86 and PL/M-286, OFFSET$OF is a built-in WORD function. For PL/M-386, OFFSET$OF is a built-in OFFSET function. It is activated by a function reference with the following form:

```
OFFSET$OF (pointer)
```

Where:

pointer	is an expression with a POINTER value.

9.8.4 The Set POINTER Bytes to Zero Variable

NIL is a built-in POINTER pseudo-variable that represents a pointer with all bytes set to zero. NIL is activated by a function reference with the following form:

```
NIL
```

The pointer value NIL points to no object. The value NIL can be assigned to a pointer to indicate, for instance, the end of a linked list.

Note that pointer values equal to NIL cannot be used to de-reference data values. For example, if a program contains the following statements:

```
DECLARE P POINTER;
DECLARE B BASED P BYTE;
P = NIL;
```

any subsequent references to B are invalid and will cause a trap.

The NIL POINTER variable also has the property that @NIL is equal to NIL.

POINTER variables can be initialized to NIL by using @NIL with INITIAL. For example:

```
DECLARE ENDOFLIST POINTER
      INITIAL (@NIL);
```

initializes ENDOFLIST with the value of NIL (i.e., all zeros). OFFSET$OF(NIL) and .NIL are also equal to zero.

NOTE

Using the NIL pointer can lead to undesirable results.

In the SMALL case, a NIL pointer is defined to be DS:0, which is not distinguishable from a valid pointer to the first variable in the data segment (DS). For example:

```
$SMALL
M:DO;
     DECLARE P POINTER INITIAL (@P);
     DECLARE B BYTE;
     B = (P=NIL);      /* will assign TRUE (0FFH) to B */
END;
```

In all other cases, the NIL pointer is defined to be 0:0, which is not distinguishable from a valid pointer to a variable located at absolute address 0. For example:

```
$LARGE
M:DO;
     DECLARE Q BYTE AT (0);
     DECLARE B BYTE;
     B + (@Q=NIL);     /* will assign TRUE (0FFH) to B */
END;
```

9.9 WORD16 Built-In Mapping for the 80386 Microprocessor

The native machine word for the 80386 microprocessor is WORD32 (a 32-bit WORD). The WORD16 control effects the semantics of some data types and built-ins as listed in Table 9-3. In PL/M-386, WORD16 keywords are mapped to the equivalent WORD32 keyword. SELECTOR, POINTER, OFFSET, (ADDRESS) are the same under both WORD32 and WORD16. WORD32 terminology is also repeated in the following table.

Table 9-3 WORD32/WORD16 Mapping for Built-ins

	WORD32	WORD16
Type conversion built-ins (casts)	BYTE HWORD WORD DWORD, QWORD CHARINT SHORTINT INTEGER	BYTE, HWORD WORD DWORD QWORD SHORTINT, CHARINT INTEGER LONGINT
8-bit built-ins	MOVB MOVRB FINDB FINDRB SKIPB SKIPRB CMPB SETB	MOVB MOVRB FINDB FINDRB SKIPB SKIPRB CMPB SETB
16-bit built-ins	MOVHW MOVRHW FINDHW FINDRHW SKIPHW SKIPRHW CMPHW SETHW	MOVW MOVRW FINDW FINDRW SKIPW SKIPRW CMPW SETW
32-bit built-ins	MOVW MOVRW FINDW FINDRW SKIPW SKIPRW CMPW SETW	MOVD MOVRD FINDD FINDRD SKIPD SKIPRD CMPD SETD

end
PL/M-386

PL/M Features Involving the Target CPU and Numeric Coprocessor 10

Tabs for
452161-001

10 FEATURES INVOLVING THE TARGET CPU AND NUMERIC COPROCESSOR CONTENTS

intel

10 FEATURES INVOLVING THE TARGET CPU AND NUMERIC COPROCESSOR

The PL/M features described in this chapter make direct or indirect use of the target microprocessor and numeric coprocessor hardware.

10.1 Microprocessor Hardware-Dependent Statements

10.1.1 The ENABLE and DISABLE Statements

These statements enable and disable the microprocessor interrupt mechanism.

The ENABLE statement has the following form:

```
ENABLE;
```

ENABLE generates an STI instruction, causing the microprocessor to enable interrupts after the next machine instruction is executed.

The DISABLE statement has the following form:

```
DISABLE;
```

DISABLE generates a CLI instruction, causing the microprocessor to disable interrupts.

10.1.2 The CAUSE$INTERRUPT Statement

The CAUSE$INTERRUPT statement causes a software interrupt to be generated. It takes the form:

CAUSE$INTERRUPT (*constant*);

Where:

constant is a whole-number constant in the range 0 to 255.

CAUSE$INTERRUPT generates an INT instruction with the constant as the interrupt type, causing the microprocessor to transfer control to the appropriate interrupt vector. Appendix G contains more information on run-time interrupt processing.

10.1.3 The HALT Statement

The HALT statement cause a microprocessor halt with interrupts enabled. The HALT statement has the form:

```
HALT;
```

It generates an STI instruction followed by an HLT instruction, causing the microprocessor to halt with interrupts enabled.

10.2 Microprocessor Hardware Flags

10.2.1 Optimization and the Hardware Flags

To produce an efficient machine-code program from a PL/M source program, PL/M compilers perform extensive optimizations of the machine code. This means that the exact sequence of machine code produced to implement a given sequence of PL/M source statements cannot be predicted.

Consequently, the state of the microprocessor hardware flags cannot be predicted for any given point in the program. For example, suppose that a source program contains the following fragment:

```
...
SUM = SUM + 250;
...
```

Where:

`SUM` is a BYTE variable.

Now, if the value of SUM before this assignment statement is greater than five, the addition will cause an overflow and the hardware CARRY flag will be set.

If there were no optimization of the machine code, this assignment statement could be followed with one of the PL/M features described in the following sections. This would ensure that the feature would operate in a certain fashion depending on whether or not the addition caused the CARRY flag to be set. However, because of the optimization, some machine code instructions could occur immediately after the addition and change the CARRY flag. It cannot be safely predicted whether this will happen or not.

NOTE

Accordingly, any PL/M feature that is dependent on the CARRY flag (or any of the other hardware flags) can cause the program to run incorrectly. These features must therefore be used with caution, and any program that uses them must be checked carefully to make sure that it operates correctly.

10.2.2 The CARRY, SIGN, ZERO, and PARITY Functions

These built-in BYTE functions return the logical values of the microprocessor hardware flags. These functions take no parameters, and are activated by function references with the following forms:

```
CARRY
ZERO
SIGN
PARITY
```

An occurrence of one of these activations (in an expression) generates a test of the corresponding condition flag. If the flag is set (= 1), a value of 0FFH is returned. If the flag is clear (= 0), a value of 0 is returned.

10.2.3 The PLUS and MINUS Operators

In addition to the arithmetic operators described in Section 5.6, PL/M has two more: PLUS and MINUS.

PLUS and MINUS perform similarly to + and −, and have the same precedence. However, PLUS sums two numbers and adds the CARRY bit to the result and MINUS subtracts two numbers and subtracts the CARRY bit from the result.

10.2.4 Carry-Rotation Functions

SCL and SCR are built-in rotation functions whose types depend on the type of the expression given as an actual parameter. They are activated by function references with the following forms:

keyword (*pattern*, *count*);

Where:

	PL/M-86	PL/M-286	PL/M-386
keyword	SCL, SCR	SCL, SCR	SCL, SCR
pattern and *count*	expressions with BYTE or WORD value — the high-order bits are dropped to produce BYTE values	expressions with BYTE or WORD value — the high-order bits are dropped to produce BYTE values	expressions with BYTE, HWORD, WORD, OFFSET, or DWORD value — the high-order bits are dropped to produce BYTE values

If the value of *count* is 0, no shift occurs.

For PL/M-86 and PL/M-286, the value of *pattern* is handled as an 8-bit or 16-bit binary quantity. For PL/M-386, the value of *pattern* is handled as an 8-bit, 16-bit, 32-bit, or 64-bit binary quantity. This quantity is rotated to the left (by SCL) or to the right (by SCR). This is similar to the ROL and ROR functions described in Chapter 9. The type of *pattern* determines the type of rotate that is performed. The number of bit positions by which the value of *pattern* is rotated is specified by *count*.

The bit rotated off one end of pattern is rotated into the CARRY flag, and the old value of CARRY is rotated to the other end of pattern. In effect, SCL and SCR perform 9-bit rotations on 8-bit values and 17-bit rotations on 16-bit values, and so on.

For example, if the value of CARRY is 0, then:

`SCL(11001010B, 2)`	returns a value of 00101001B and CARRY is set to 1
`SCR(11001010B, 1)`	returns a value of 01100101B and CARRY remains 0

10.2.5 The Decimal Adjust Function

DEC is a built-in BYTE function that performs a decimal adjust operation on the actual parameter value and returns the result of this operation. For PL/M-86 and PL/M-286, DEC uses the value of the hardware CARRY flag internally. For PL/M-386, DEC uses the value of the hardware AUXILIARY CARRY flag internally. It is activated by a function reference with the following form:

DEC (*expression*);

Where:

expression is converted, if necessary, to a BYTE value.

10.3 Microprocessor Hardware Registers

10.3.1 The Flags Register Access Variable

FLAGS is a built-in WORD variable that provides access to the microprocessor's hardware flags register (see Figure 10-1). The hardware flags register contains the hardware flags that are altered by the execution of various instructions. The hardware flags register for the 8086 and the 80286 microprocessors is 16-bit. The hardware flags register for the 80386 microprocessor is 32-bit.

The FLAGS register is assigned to change the setting of the various flags. It can also be read to determine the current flag settings.

For more information on setting the hardware register flags, see the appropriate programmer's reference manual.

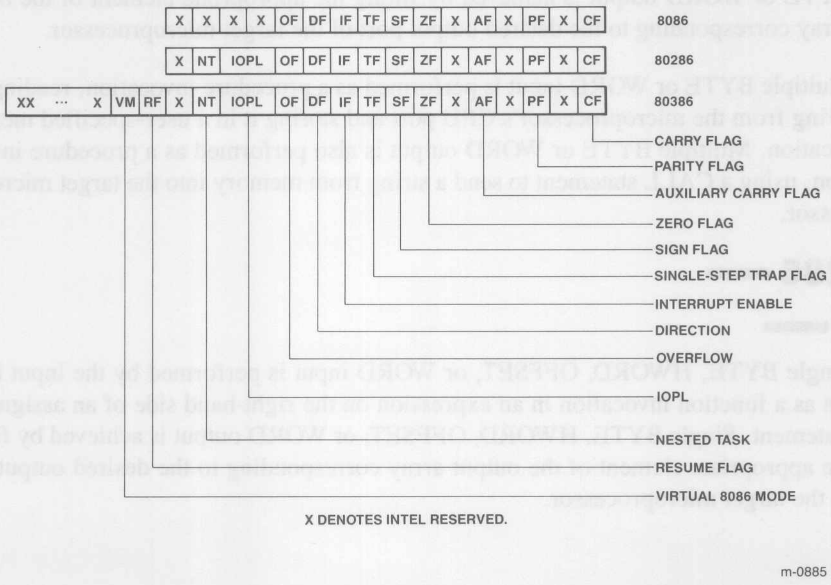


Figure 10-1 The Hardware Flags Register

10.3.2 The STACKPTR and STACKBASE Variables

For PL/M-86 and PL/M-286, STACKPTR and STACKBASE are WORD variables. For PL/M-386, STACKPTR is an OFFSET variable and STACKBASE is a SELECTOR variable. They provide access to the microprocessor's hardware stack pointer and stack base registers.

When setting these registers (that is, using STACKPTR or STACKBASE on the left side of an assignment), care must be exercised because this takes control of the stack away from the compiler. Thus, the compile-time checks on stack overflow and assumptions by the compiler about the run-time status of the stack may be invalid.

10.4 Microprocessor Hardware I/O

PL/M-86/286

Single BYTE or WORD input is performed by the input built-ins as a function invocation in an expression on the right-hand side of an assignment statement. Single BYTE or WORD output is achieved by filling the appropriate element of the output array corresponding to the desired output port of the target microprocessor.

Multiple BYTE or WORD input is performed as a procedure invocation, reading in a string from the microprocessor's CPU port and storing it in a user-specified memory location. Multiple BYTE or WORD output is also performed as a procedure invocation, using a CALL statement to send a string from memory into the target microprocessor.

end PL/M-86/286

PL/M-386

Single BYTE, HWORD, OFFSET, or WORD input is performed by the input built-ins as a function invocation in an expression on the right-hand side of an assignment statement. Single BYTE, HWORD, OFFSET, or WORD output is achieved by filling the appropriate element of the output array corresponding to the desired output port of the target microprocessor.

Multiple BYTE, HWORD, OFFSET, or WORD input is performed as a procedure invocation, reading in a string from the microprocessor's CPU port and storing it in a user-specified memory location. Multiple BYTE, HWORD, OFFSET, or WORD output is also performed as a procedure invocation, using a CALL statement to send a string from memory into the target microprocessor port.

10.4.1 The Find Value in Input Port Function

The following built-in functions return the values in the specified input port. They are activated by function references with the form:

keyword (*expression*);

Where:

	PL/M-86	PL/M-286	PL/M-386*
keyword	INPUT, INWORD	INPUT, INWORD	INPUT, INHWORD, INWORD
expression	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD or WORD value

*See also Section 10.4.3.

The value of *expression* specifies one of the input ports of the target microprocessor.

The value returned by the *keyword* is the *expression* quantity found in the specified input port.

PL/M-386 also has an INDWORD function when the WORD16 control is used.

10.4.2 The Access Output Port Array

For PL/M-86 and PL/M-286, OUTPUT and OUTWORD are built-in BYTE and WORD arrays, respectively. For PL/M-386, OUTPUT, OUTHWORD, and

OUTWORD are built-in BYTE, HWORD, and WORD arrays, respectively. They are activated by a function reference with the following form:

keyword (expression);

Where:

	PL/M-86	PL/M-286	PL/M-386*
keyword	OUTPUT, OUTWORD	OUTPUT, OUTWORD	OUTPUT, OUTHWORD, OUTWORD
expression	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, or WORD value

*See also Section 10.4.4.

These functions can access any port from 0 to 65,556, corresponding to the number of output ports on the target CPU. References to these arrays cause the specified *expression* quantity to be latched to the specified hardware output port.

A reference to *keyword* is legal only as the left part of an assignment statement or embedded assignment. For PL/M-86 and PL/M-286, the right-hand side of the assignment must have a BYTE or WORD value. For PL/M-386, the right-hand side of the assignment must have a BYTE, HWORD, or WORD value.

Assignment to an element of OUTPUT places the BYTE value of the expression on the right side of the assignment into the corresponding output port. (Since OUTPUT is a BYTE built-in, the value of the expression is converted automatically to a type BYTE if necessary.)

Assignment to an element of OUTWORD places the WORD (additionally, for PL/M-386, OFFSET) value of the expression on the right side of the assignment into the corresponding output port.

For PL/M-386, assignment to an element of OUTHWORD places the HWORD value of the expression on the right side of the assignment into the corresponding output port.

PL/M-386 also has an OUTDWORD built-in when the WORD16 control is used (see Section 10.8).

10.4.3 The Read and Store String Procedure

The read and store string procedures are built-in. For PL/M-86 and PL/M-286, these built-ins read the BYTE or WORD string values latched to the specified hardware input port. For PL/M-386, these built-ins read the BYTE, HWORD, OFFSET, or WORD string values latched to the specified hardware input port. The read values, of the length specified by *count*, are then stored at the location specified by *destination*. These procedures are activated by a CALL statement with the following form:

CALL *keyword* (*port, destination, count*);

Where:

	PL/M-86*	PL/M-286	PL/M-386**
keyword	BLOCKINPUT, BLOCKINWORD	BLOCKINPUT, BLOCKINWORD	BLOCKINPUT, BLOCKINHWORD, BLOCKINWORD
port	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE or HWORD value
destination	expression with POINTER value	expression with POINTER value	expression with POINTER value
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

*To use this procedure in PL/M-86, the MOD186 control must be used (see Section 11.1.1).
**See Section 10.4.1.

The *keyword* specifies the type of string found in the specified input port. The value of *port* specifies one of the input ports of the CPU. The *destination* specifies the location (in memory) at which to store the string. The value of *count* specifies the length of the string.

PL/M-386 also has a BLOCKINDWORD procedure when the WORD16 control is used (see Section 10.8).

10.4.4 The Write String Procedure

The write string procedures are built-in procedures. For PL/M-86 and PL/M-286, these built-ins write a BYTE or WORD string to the specified hardware output port. For PL/M-386, these built-ins write a BYTE, HWORD, OFFSET, or WORD string to the specified output hardware port. These built-ins are activated by a CALL statement with the following form:

CALL *keyword* (*port, source, count*);

Where:

	PL/M-86*	PL/M-286	PL/M-386**
keyword	BLOCKOUTPUT, BLOCKOUTWORD	BLOCKOUTPUT, BLOCKOUTWORD	BLOCKOUTPUT, BLOCKOUTHWORD, BLOCKOUTWORD
port	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE or HWORD value
source	expression with POINTER value	expression with POINTER value	expression with POINTER value
count	expression with BYTE or WORD value	expression with BYTE or WORD value	expression with BYTE, HWORD, OFFSET, or WORD value

*To use this procedure in PL/M-86, the MOD186 control must be used (see Section 11.1.1).
**See Section 10.4.2.

The *keyword* specifies the type of string. The value of *port* specifies one of the output ports of the microprocessor CPU. The *source* value specifies the location (in memory) where the string is currently stored. The value of *count* specifies the string length.

PL/M-386 also has a BLOCKOUTDWORD procedure when the WORD16 control is used (see Section 10.8).

10.5 The Hardware Protection Model

The 80286 and the 80386 microprocessors' protection mechanism provides up to four privilege levels within each task. The highest privilege level (level 0) is reserved for the operating system kernel. Below the kernel level, systems can be configured to include a system service level (level 1), an applications service level (level 2), and an application program level (level 3).

The following hardware protection built-in procedures and variables allow access to the protection architecture of the 80286 and the 80386 microprocessors.

10.5.1 The Task Register

10.5.1.1 The TASK$REGISTER Variable

TASK$REGISTER is a built-in SELECTOR variable that provides access to the task state register. This register points to a task state segment for the currently executing task. The format of the task register is:

INDEX				TI	RPL	
31	PL/M-286	...	19	18	17	16
15	PL/M-386	...	3	2	1	0

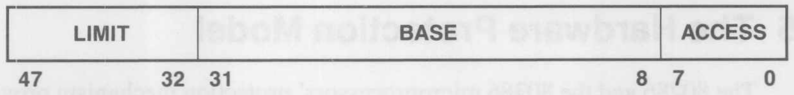
Values are assigned to TASK$REGISTER to reset the task state segment for the current task or to enter the protected mode of the microprocessor. However, the selector stored in TASK$REGISTER must point to a valid task state segment. Note that values can only be assigned to TASK$REGISTER if the program is executed in protection mode at level 0.

TASK$REGISTER can also be read to determine the task state segment of the currently executing task.

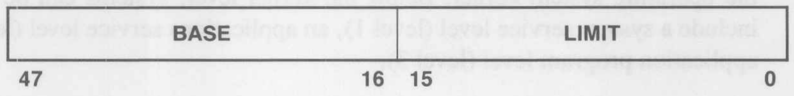
10.5.2 The Global Descriptor Table Register

The global descriptor table register (GDTR) is a system-wide register used for protected virtual address mode. The GDTR describes a memory area that contains an array of descriptors for the global address space. The register occupies 6 bytes.

Its format for the 80286 microprocessor is as follows:



Its format for the 80386 microprocessor is as follows:



LIMIT — size of the GDT segment (up to 64K bytes)

BASE — physical memory base address of the GDT segment

ACCESS — access control byte

10.5.2.1 The SAVE$GLOBAL$TABLE Procedure

SAVE$GLOBAL$TABLE is a built-in procedure. It is activated by a CALL statement with the form:

```
CALL SAVE$GLOBAL$TABLE (location);
```

Where:

location	is an expression with a POINTER value.

SAVE$GLOBAL$TABLE saves the contents of the hardware global descriptor table register in the 6-byte save area pointed to by *location*.

Specific to the 80286 microprocessor, the memory value of the access byte is undefined after a call to SAVE$GLOBAL$TABLE.

10.5.2.2 The RESTORE$GLOBAL$TABLE Procedure

RESTORE$GLOBAL$TABLE is a built-in procedure. It is activated by a CALL statement with the form:

```
CALL RESTORE$GLOBAL$TABLE (location);
```

Where:

location	is an expression with a POINTER value.

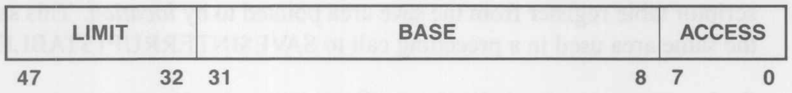
RESTORE$GLOBAL$TABLE restores the contents of the hardware global descriptor table register from the save area pointed to by *location*. This save area can be the same area used in a preceding call to SAVE$GLOBAL$STATUS.

SAVE$GLOBAL$TABLE saves the value of the GDTR in a 6-byte memory area. RESTORE$GLOBAL$TABLE restores the value of the GDTR.

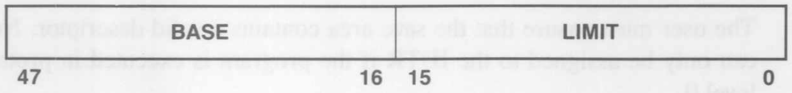
10.5.3 The Interrupt Descriptor Table Register

The interrupt descriptor table register (IDTR) is a system-wide register that is used for interrupt processor management. The IDTR describes a segment that contains the linear base address and the size of the interrupt descriptor table (IDT), and a segment containing an array of gate descriptors for the interrupt handlers. The register occupies 6 bytes.

For the 80286 microprocessor, it has the following format:



For the 80386 microprocessor, it has the following format:



LIMIT — size of the segment (up to 64K bytes)

BASE — physical memory base address of the IDT segment

ACCESS — access control byte

10.5.3.1 The SAVE$INTERRUPT$TABLE Procedure

SAVE$INTERRUPT$TABLE is a built-in procedure that is activated by a CALL statement with the following form:

```
CALL SAVE$INTERRUPT$TABLE (location);
```

Where:

location is an expression with a POINTER value.

SAVE$INTERRUPT$TABLE saves the contents of the hardware interrupt descriptor table register in the 6-byte save area pointed to by *location*.

10.5.3.2 The RESTORE$INTERRUPT$TABLE Procedure

RESTORE$INTERRUPT$TABLE is a built-in procedure that is activated by a CALL statement with the following form:

```
CALL RESTORE$INTERRUPT$TABLE (location);
```

Where:

location is an expression with a POINTER value.

RESTORE$INTERRUPT$TABLE restores the contents of the hardware interrupt descriptor table register from the save area pointed to by *location*. This save area can be the same area used in a preceding call to SAVE$INTERRUPT$TABLE.

A descriptor can be built that will initialize the interrupt processor operations. RESTORE$GLOBAL$STATUS can then be called with a pointer to this descriptor.

The user must ensure that the save area contains a valid descriptor. Note that values can only be assigned to the IDTR if the program is executed in protection mode at level 0.

10.5.4 The Local Descriptor Table Register

10.5.4.1 The LOCAL$TABLE Variable

LOCAL$TABLE is a built-in SELECTOR variable that provides access to the local descriptor table register (LDTR). The format of the register is a selector pointing to an LDT in the GDT. The use of the local descriptor table is like the use of the GDTR, except that it defines the local address space.

By assigning a value to LOCAL$TABLE, the local address space of the current task is altered. If a task switch occurs, the new contents are not saved in the task state segment. (To ensure proper operation, interrupts must be disabled.)

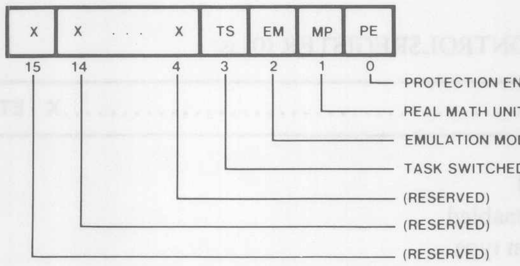
LOCAL$TABLE can be read to determine the current active descriptor array segment for the current task.

The user must ensure that the selector in LOCAL$TABLE points to a valid descriptor segment. Note that values can only be assigned to the LDTR when the program is executed in protection mode at level 0.

10.5.5 The Machine Status Register

10.5.5.1 The MACHINE$STATUS Variable

For PL/M-286, MACHINE$STATUS is a built-in WORD variable. For PL/M-386, MACHINE$STATUS is a built-in HWORD variable. MACHINE$STATUS provides access to the machine status word (MSW). The MSW register defines the current status of the processor protection model and the real math unit support. The format of MACHINE$STATUS is:



121945-3

PL/M-286/386
(continued)

MACHINE$STATUS enables access to the protected mode of the microprocessor. When a value is assigned to this register, the compiler generates a short jump to the next instruction to clear the instruction queue. (Note, however, that values can only be assigned to MACHINE$STATUS if the program is executed in protection mode at level 0.)

NOTE

Once the protection mode has been accessed, the only way to return to real address mode is by executing a hardware RESET.

The contents of MACHINE$STATUS can also be read to determine the current status of various system components.

end
PL/M-286/386

PL/M-386

10.5.5.2 The CONTROL$REGISTER, DEBUG$REGISTER, and TEST$REGISTER Built-In Arrays

The CONTROL$REGISTER is a built-in WORD array that provides access to the 80386 microprocessor's 32-bit control registers that define the current status of the processor and contain page table and page fault information.

The format of CONTROL$REGISTER (0) is:

PG	X..X	ET	TS	EM	MP	PE

X	reserved
PG	paging enabled
ET	extension type
TS	task switched
EM	emulation mode
MP	real math unit (numeric coprocessor) present
PE	protection enable

MSW is contained in the low-order 16 bits of CONTROL$REGISTER (0). However, assigning a value to the MACHINE$STATUS built-in does not change the ET (extension type) bit.

CONTROL$REGISTER (2) contains the 32-bit linear address that caused the last detected page fault.

CONTROL$REGISTER (3) contains the physical page base address for the first level of the page table structure. This address is in the high 20 bits (bits 12 to 31) of CONTROL$REGISTER (3). The lower 12 bits are ignored when assigning to CONTROL$REGISTER (3) and are undefined when reading CONTROL$REGISTER (3). Note that the control registers are accessible only during execution at protected mode level 0. Also note that CONTROL$REGISTER (1) is not accessible.

The DEBUG$REGISTER built-in WORD array provides access to six of the eight 32-bit debug registers; DEBUG$REGISTER(4) and DEBUG$REGISTER(5) are not accessible. The debug registers are accessible only during execution at protected mode level 0.

TEST$REGISTER is a built-in WORD array that provides access to the 32-bit test registers of the 80386 microprocessor. Of these test registers, only TEST $REGISTER (6) and TEST$REGISTER (7) are accessible; these registers are accessible only when executing in protection mode at level 0.

10.5.5.3 The CLEAR$TASK$SWITCHED$FLAG Procedure

CLEAR$TASK$SWITCHED$FLAG is a built-in procedure that is activated by a CALL statement with the form:

```
CALL CLEAR$TASK$SWITCHED$FLAG;
```

This procedure is used to clear the task switched flag in the machine status word. The processor sets the task switched flag every time a task switch occurs. It can be used to manage the sharing of the real math coprocessor.

CLEAR$TASK$SWITCHED$FLAG can only be called when the program is executed in protection mode at level 0.

10.5.6 Segment Information

10.5.6.1 The GET$ACCESS$RIGHTS Function

GET$ACCESS$RIGHTS is a built-in WORD function; it is activated by a function reference with the form:

GET$ACCESS$RIGHTS (*selector*);

Where:

selector	is an expression with a SELECTOR value.

If the segment pointed to by *selector* is visible at the current privilege level, then the hardware ZERO flag is set and a WORD value returned. If the segment is not visible, or if it is of the wrong type, the hardware ZERO flag is reset, and the value returned is undefined.

NOTE

The setting of the ZERO flag is guaranteed only if it is tested immediately, before being altered by another operation. (For example, if the value of the function is assigned to an array element indexed by an expression, the value of the ZERO flag may be incorrect.)

Specific to the 80386 microprocessor, the format of the return value is:

0 0	G	0	0	AVL	X X	ACCESS	0 0
31 24	23	22	21	20	19 16	15 8	7 0

X	reserved
G	granularity bit
AVL	available for software use
ACCESS	access rights byte

The following example illustrates how the GET$ACCESS$RIGHTS function can be used:

```
DECLARE RIGHTS WORD;
DECLARE SEGMENT SELECTOR;

          RIGHTS = GET$ACCESS$RIGHTS (SEGMENT);
          IF ZERO THEN
               /* The segment pointed to by SEGMENT is visible */
                                /* and RIGHTS contains the proper access */
                                                          /* rights to it. */
          ELSE
                    /* SEGMENT is not visible and the contents of */
                                                  /* RIGHTS is undefined. */
```

10.5.6.2 The GET$SEGMENT$LIMIT Function

For PL/M-286, GET$SEGMENT$LIMIT is a built-in WORD function. For PL/M-386, GET$SEGMENT$LIMIT is a built-in OFFSET function. GET-$SEGMENT$LIMIT is activated by a function call of the form:

GET$SEGMENT$LIMIT (*selector*);

Where:

selector is an expression with a SELECTOR value.

If the segment pointed to by *selector* is visible at the current protection level, then the hardware ZERO flag is set and the value returned by GET$SEGMENT$LIMIT is the size of the segment. Otherwise, if the segment is not visible, the ZERO flag is reset and the value returned is undefined.

Set the ZERO flag with caution. See the note in Section 10.5.6.1.

The following example (for PL/M-286) illustrates how the GET$SEGMENT$LIMIT function can be used:

```
DECLARE LIMITS WORD;
DECLARE SEGMENT SELECTOR;

LIMITS = GET$SEGMENT$LIMIT (SEGMENT);
IF ZERO THEN
               /* The segment pointed to by SEGMENT is visible */
                                   /* and LIMITS contains its proper size. */
ELSE
                    /* SEGMENT is not visible and the contents of */
                                                  /* LIMITS is undefined. */
```

10.5.7 Segment Accessibility

It is sometimes helpful to know if the segment pointed to by a selector is readable or writable from the current address space. This becomes particularly important when the selector is a parameter that is passed to the current task.

If an attempt is made to access a segment that is inaccessible, an interrupt will occur. To avoid this interrupt, segment readability and writability can be tested before the segment is accessed.

10.5.7.1 The SEGMENT$READABLE Function

SEGMENT$READABLE is a built-in BYTE function. It is activated by a function reference with the form:

SEGMENT$READABLE (*selector*);

Where:

selector is an expression with a SELECTOR value.

SEGMENT$READABLE returns a value of TRUE (0FFH) if the segment pointed to by *selector* is reachable and readable from the current privilege level; FALSE (0), if it is not.

10.5.7.2 The SEGMENT$WRITABLE Function

SEGMENT$WRITABLE is a built-in BYTE function. It is activated by a function reference with the form:

SEGMENT$WRITABLE (*selector*);

Where:

selector is an expression with a SELECTOR value.

SEGMENT$WRITABLE returns a value of TRUE (0FFH) if the segment pointed to by *selector* is reachable and writable from the current privilege level; FALSE (0), if it is not.

10.5.8 Adjusting the Requested Privilege Level

10.5.8.1 The ADJUST$RPL Function

ADJUST$RPL is a built-in SELECTOR function that returns the argument of the adjusted requested privilege level (RPL). It is activated by a function reference with the form:

ADJUST$RPL (*selector*);

Where:

selector is an expression with a SELECTOR value.

If the requested privilege level (RPL) field of the argument selector is less than the RPL field of the code segment selector for the routine calling the procedure that invoked ADJUST$RPL, then the hardware ZERO flag is set and the value returned is the argument of an adjusted RPL field. Otherwise, the ZERO flag is reset, and the value returned is the original value of the argument.

Setting the ZERO flag should be used with caution, see the note in Section 10.5.6.1.

The following example illustrates how the ADJUST$RPL function can be used:

```
P: PROCEDURE (SEGMENT);
      DECLARE SEGMENT SELECTOR;

      SEGMENT = ADJUST$RPL (SEGMENT);
      IF ZERO THEN
       /* The RPL of SEGMENT was less than the RPL of */
        /* the routine that called P; SEGMENT now has */
                                /* the RPL of the caller. */
      ELSE
      /* The RPL of SEGMENT was not less than the RPL */
/* of the routine that called P; SEGMENT is unchanged */

END P;
```

10.6 The REAL Math Facility

REAL math support for PL/M is provided by the numeric coprocessor. In relation to the program, the REAL math facility consists of the following:

- The REAL stack, used to hold operands and results during REAL operations.
- The REAL error byte (see Figure 10-2), consisting of seven exception flags initialized to all 0s. (The reserved bit is set to 1 by the numeric coprocessor.)

  The first six bits in this byte correspond to the possible errors that can arise during REAL operations (see Appendix G). When an error occurs, the facility sets the corresponding bit to 1. A program can invoke a built-in procedure (described in Section 10.7) that reads and clears the REAL error byte.

  The exception/error categories are discussed in Appendix G.
- The REAL mode word (see Figure 10-3), consisting of 16 bits initialized to 03FFH (or 7FFH for the 80387 numeric coprocessor).

  1. Bits 0-7 determine whether the corresponding error condition is to be handled with the default recovery (described next) or with the programmer-supplied exception procedure (see Appendix G for details on writing these). When the bit is 1, the default is used; when it is 0, the user routine is used. In either case, the facility records the error by setting the corresponding bit of the REAL error byte. For most uses, the default recovery is appropriate and less work.

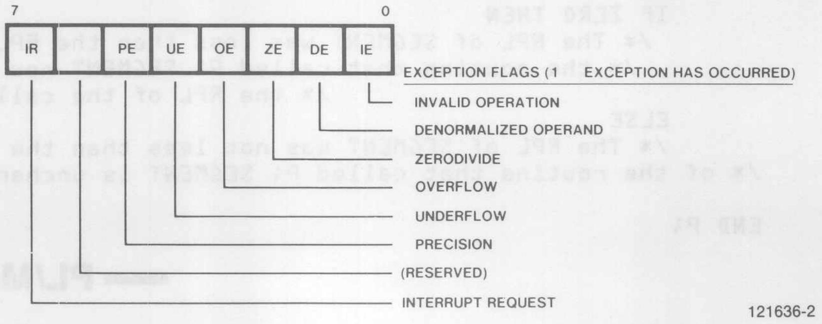
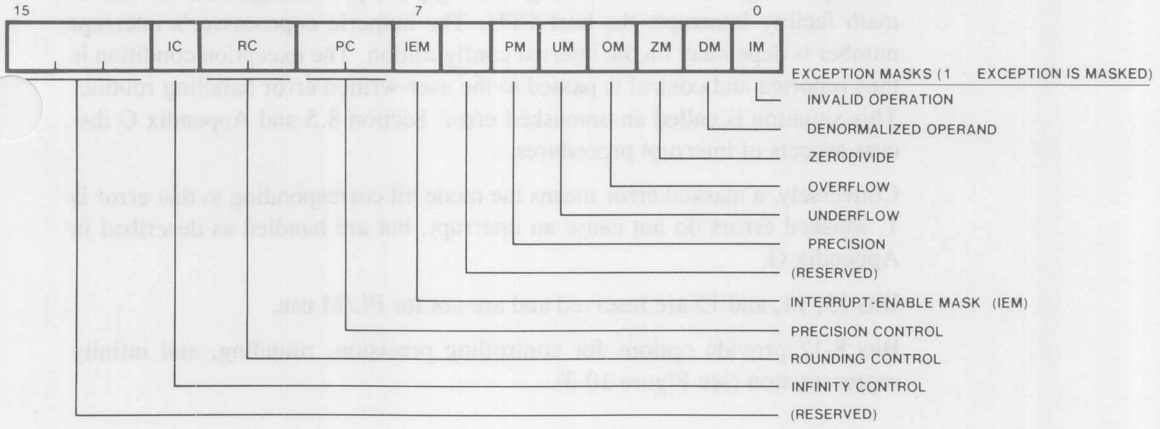


Figure 10-2 The REAL Error Byte



Interrupt-Enable Mask:

0	Interrupts Enabled
1	Interrupts Disabled (Masked)

Precision Control:

00	24 bits
01	(reserved)
10	53 bits
11	64 bits

Rounding Control:

00	Round to Nearest or Even
01	Round Down (toward)
10	Round Up (toward .)
11	Chop (Truncate Toward Zero)

Infinity Control:

0	Projective
1	Affine

121636-3

Figure 10-3 The REAL Mode Word

This mode word is often called a mask; that is, it lets some signals through (to interrupt processing), but not others. If one of the bits 0-5 is a 0, the corresponding error is said to be unmasked (see Section 10.7 on how to set the mode word).

If the interrupt is enabled (IEM = 0), one of the masked bits is 0, and the corresponding error occurs during floating point processing, then the REAL math facility interrupts the host CPU. The numeric coprocessor's interrupt number is dependent on the internal configuration. The exception condition is thus reported and control is passed to the user-written error handling routine. This situation is called an unmasked error. Section 8.5 and Appendix G discuss aspects of interrupt procedures.

Conversely, a masked error means the mode bit corresponding to that error is 1. Masked errors do not cause an interrupt, but are handled as described in Appendix G.

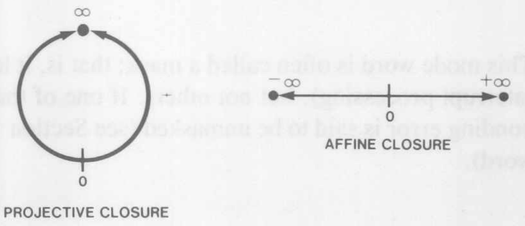
Bits 13, 14, and 15 are reserved and are not for PL/M use.

Bits 8-12 provide options for controlling precision, rounding, and infinity representation (see Figure 10-3).

2. All intermediate results are held in an internal format of 64-bit precision. The most-significant 24 bits of the final result are returned (plus sign and 7-bit exponent) as the PL/M answer, and rounded, if needed, according to the user-specified control. The default precision setting preserves extended precision and operates slightly faster than the other.

3. Rounding introduces an error of less than one unit in the last place to which the result was rounded. Statistically, the default provides the most accurate and unbiased estimate of the true result (i.e., the 64-bit result). In all rounding modes except round down, subtracting a number from itself yields +0; round down yields −0.

4.



121945-4

10.7 Built-Ins Supporting the REAL Math Unit

10.7.1 The INIT$REAL$MATH$UNIT Procedure

INIT$REAL$MATH$UNIT is a built-in untyped procedure activated by a CALL statement, as follows:

```
CALL INIT$REAL$MATH$UNIT;
```

This call is required as the first access to the REAL math facility (the numeric coprocessor).

This call initializes the REAL math unit for subsequent operations. This includes setting a default value into the control (REAL mode) word, namely 03FFH in hexadecimal or 0000001111111111 in binary. This setting masks all exceptions and interrupts, sets precision to 64 bits, and sets the rounding mode to even. This means no interrupts will occur from the REAL math facility regardless of what errors are detected.

Procedures activated after this call has taken effect do not need to do such initialization.

10.7.2 The SET$REAL$MODE Procedure

This procedure should only be invoked to change the default mode word (for example, to unmask the invalid exception).

SET$REAL$MODE is a built-in untyped procedure, activated by a CALL statement with the following form:

CALL SET$REAL$MODE (*modeword*);

Where:

	PL/M-86	PL/M-286	PL/M-386
modeword	expression with WORD value	expression with WORD value	expression with HWORD value

The value of *modeword* becomes the new contents of the REAL mode word (see Figure 10-3). The suggested value for *modeword* is 033EH, (0000001100111110 in binary). This value provides maximum precision, default rounding, and masked handling of all exception conditions except invalid, which can alert the user to errors of initialization or stack usage. See Appendix G for facts and references on writing an interrupt handling procedure.

10.7.3 The GET$REAL$ERROR Function

GET$REAL$ERROR is a built-in BYTE function activated by a function reference with the following form:

```
GET$REAL$ERROR
```

The BYTE value returned is the current contents of the REAL error byte (see Figure 10-2). This function also clears the error byte in the REAL math facility.

10.7.4 Saving and Restoring REAL Status

If an interrupt procedure performs any floating-point operation, it will change the REAL status. If such an interrupt procedure is activated during a floating-point operation, the program will be unable to continue the interrupted operation correctly after returning from the interrupted procedure. Therefore, it is first necessary for any interrupt procedure that performs a floating-point operation to save the REAL status and subsequently restore it before returning. The built-in procedures SAVE$REAL$STATUS and RESTORE$REAL$STATUS make this possible. SAVE$REAL$STATUS also initializes the numeric coprocessor.

Additionally, these procedures can be used in a multi-tasking environment where a running task using the numeric coprocessor can be preempted by another task that also uses the numeric coprocessor. The preempting task must call SAVE$REAL-$STATUS before it executes any statements that affect the numeric coprocessor, that is, before calling SET$REAL$MODE and before any arithmetic or assignment of REALs (other than GET$REAL$ERROR, if needed).

New vectors will be required for the interrupt handlers appropriate to each new task (e.g., to handle unmasked exception conditions). These vectors must be initialized by the operating system.

After its processing is complete and it is ready to terminate, the preempting task must call RESTORE$REAL$STATUS to reload the state information that applied at the time the former running task was preempted. This enables that task to resume execution from the point where it relinquished control.

NOTE

REAL functions without REAL parameters should not call GET$REAL$ERRORS or SAVE$REAL$STATUS before executing at least one floating-point instruction. To do so may result in loss of processor synchronization.

10.7.4.1 The SAVE$REAL$STATUS Procedure

SAVE$REAL$STATUS is a built-in untyped procedure activated by a CALL statement with the form:

```
CALL SAVE$REAL$STATUS (location);
```

Where:

location	is a pointer to a memory area (94 bytes for PL/M-86 and PL/M-286 and 108 bytes for PL/M-386) where the REAL status information will be saved.

The REAL status is saved at the specified location, and the REAL stack and error bytes are reinitialized.

If the state of the REAL math unit is unknown to this procedure when it is called, as in the case previously mentioned for preempting tasks, then an initialization will destroy existing error flags, masks, and control settings. To avoid this, the appropriate action (except for error-recovery routines, discussed in Appendix G) is to issue:

```
CALL SAVE$REAL$STATUS (@location_1);
```

before any REAL math usage, and

```
CALL RESTORE$REAL$STATUS (@location_1);
```

prior to the procedure's return. The save automatically reinitializes the math unit and the error byte.

This protects the status of preempted tasks or prior procedures and establishes a known initialization state for the current procedure's actions. The microprocessor interrupts are disabled during the save.

NOTE

The microprocessor must be able to acknowledge numeric coprocessor interrupts or loss of synchronization occurs.

10.7.4.2 The RESTORE$REAL$STATUS Procedure

RESTORE$REAL$STATUS is a built-in untyped procedure activated by a CALL statement with the form:

```
CALL RESTORE$REAL$STATUS (location);
```

Where:

location is a pointer to a memory area where the REAL status information was previously saved by a call to the SAVE$REAL$STATUS procedure.

This procedure should be called prior to returning from an interrupt procedure where the real math unit's status was saved using SAVE$REAL $STATUS.

PL/M-286/386

10.7.5 Interrupt Processing

10.7.5.1 The WAIT$FOR$INTERRUPT Procedure

WAIT$FOR$INTERRUPT is a built-in procedure that is activated by a CALL statement with the form:

CALL WAIT$FOR$INTERRUPT;

This procedure is used to generate an IRET instruction in a nested interrupt task; if it is used elsewhere, the results are undefined. The IRET instruction causes the microprocessor to perform a task switch, saving the status of the outgoing task in its TSS. The next time the interrupt task is activated, execution will begin at the instruction immediately following the IRET, with all the registers unchanged.

The following example illustrates how the WAIT$FOR$INTERRUPT procedure can be used:

```
NEW$INTERRUPT:
      CALL INITIALIZE$INTERRUPT$LIST;
                            /* Start of a list of interrupts */
      DO WHILE 1;
            CALL WAIT$FOR$INTERRUPT;
                  /* Wait for next interrupt within list */
            CALL PROCESS$INTERRUPT;
            IF END$OF$INTERRUPT$LIST THEN DO;
                  CALL WAIT$FOR$INTERRUPT;
           /* Wait for start of next interrupt sequence */
                  GOTO NEW$INTERRUPT;
            END;
      END;
```

end
PL/M-286/386

10.8 WORD16 Mapping for Built-Ins

Table 10-1 lists the WORD16 terminology for the built-ins described in this chapter that specify particular data types. In PL/M-386, WORD16 keywords are mapped to the equivalent WORD32 keyword. The WORD32 terminology is repeated in this table.

Table 10-1 WORD32/WORD16 Mapping for Built-ins

WORD32	WORD16
INPUT	INPUT
OUTPUT	OUTPUT
BLOCKINPUT	BLOCKINPUT
BLOCKOUTPUT	BLOCKOUTPUT
INHWORD	INWORD
OUTHWORD	OUTWORD
BLOCKINHWORD	BLOCKINWORD
BLOCKOUTHWORD	BLOCKOUTWORD
INWORD	INDWORD*
OUTWORD	OUTDWORD*
BLOCKINWORD	BLOCKINDWORD*
BLOCKOUTWORD	BLOCKOUTDWORD*

*The WORD16 control must be used.

Compiler Controls

11

Tabs for
452161-001

11 COMPILER CONTROLS CONTENTS

11 COMPILER CONTROLS

11.1 Introduction to Compiler Controls

You can control the operation of the PL/M compilers by using the compiler controls described in this chapter. You can use controls in the command that invokes the compiler, or as control lines in the source input file.

A control line contains a dollar sign ($) in the left margin. Normally, the left margin is set at column one, but you can change this with the LEFTMARGIN control. Control lines allow selective control over sections of the program. For example, it may be desirable to suppress the listing for certain sections of the program, or to cause page ejects at certain places.

A line in a source file is considered to be a control line by the compiler if there is a dollar sign in the left margin, even if the dollar sign appears to be part of a PL/M comment or character string constant. Control lines within the source code must begin with a dollar sign and can contain one or more controls, each separated by at least one blank. Only the left margin column of a control line should contain a dollar sign.

The following are examples of control lines:

```
    $NOCODE XREF
    $EJECT CODE
```

There are two types of controls: primary and general. Primary controls must occur either in the invocation command or in a control line that precedes the first noncontrol line of the source file. Primary controls cannot be changed within a module. General controls can occur either in the invocation command or in a control line anywhere in the source input, and can be changed freely within a module. Certain controls can be negated by prefacing the control word with a NO. The control descriptions in this chapter indicate that option by showing both options in the headings.

Many controls are available, but a set of defaults is built into the compilers. The controls are summarized in alphabetic order in Table 11-1. Controls that are specific to PL/M-86 and -386 are summarized in Tables 11-1A and 11-1B, respectively.

A control consists of a control-name and, in some cases, a parameter. Parameters in control lines must be enclosed in parentheses. Enclosing control parameters on the invocation line in parentheses may be illegal, depending on the host operating system.

This chapter is organized in the following manner:

- Controls, default settings, abbreviations, and effects are listed in Table 11-1.
- Compiler controls are categorized and an overview for each of the categories is provided.
- Compiler control descriptions are provided in alphabetical order, as listed in Table 11-1. For example, the NOSYMBOLS description is located with the SYMBOLS description.
- A sample program listing is provided with a description of the listing.

Table 11-1 Compiler Controls

Control	Default	Abbrev.	Effect
CODE NOCODE	NOCODE	CO NOCO	Enables or disables listing of pseudo-assembly code.
COND NOCOND	COND	none none	Determines whether text skipped during compilation appears in the listing.
*DEBUG *NODEBUG	NODEBUG	DB NODB	Generates debug records in the object module.
EJECT	paging is automatic	EJ	Forces a new print page.

IF
ELSEIF
ELSE
ENDIF
	not applicable	none	Enables the conditional compilation capability by testing for conditions that use the value of compile-time switches.

INCLUDE	not applicable	IC	Includes other source files as input to the compiler.
*INTERFACE	none	ITF	Enables calls to other high-level languages and to source code translators.
LEFTMARGIN	LEFTMARGIN(1)	LM	Specifies that only input beginning at position *n* should be processed by the compiler.
LIST NOLIST	LIST	LI NOLI	Enables or disables listing of source program.

* Denotes primary control

Table 11-1 Compiler Controls (continued)

Control	Default	Abbrev.	Effect
*OBJECT *NOOBJECT	OBJECT (*source*.OBJ)	OJ NOOJ	Specifies a filename for an object module, or prevents creation of an object module.
*OPTIMIZE	OPTIMIZE(1)	OT	Determines the optimization level during code generation.
OVERFLOW NOOVERFLOW	NOOVERFLOW	OV NOOV	Enables or disables overflow detection during signed arithmetic.
*PAGELENGTH	PAGELENGTH(60)	PL	Specifies the maximum number of lines per page.
*PAGEWIDTH	PAGEWIDTH(120)	PW	Specifies the maximum number of characters per line.
PAGING NOPAGING	PAGING	PI NOPI	Specifies whether the program listing should be page formatted with a heading that identifies the compiler and page number. A user-specified title can also be included (see TITLE).
PRINT NOPRINT	PRINT	PR NOPR	Enables or disables printed output, or selects the device or file to receive the printed output.

* Denotes primary control

Table 11-1 Compiler Controls (continued)

Control	Default	Abbrev.	Effect
*RAM *ROM	**RAM	none	Specifying RAM places the CONSTANT section within the DATA segment in all segmentation. Specifying ROM places constants in the CODE segment.
SAVE RESTORE	none	SA RS	Enables the settings of certain controls to be saved on the stack and restores the control settings after the included file.
SET RESET	RESET(0)	none	Controls the value of switches. SET establishes a value. RESET restores the value to 0.
*SMALL *COMPACT *MEDIUM *LARGE	SMALL (PL/M-86 and PL/M-386) LARGE (PL/M-386)	SM CP MD LA	Determines the segmentation model.
SUBTITLE	no subtitle	ST	Puts a subtitle on each page of printed output and causes a page eject.
*SYMBOLS *NOSYMBOLS	NOSYMBOLS	SB NOSB	Specifies to the compiler whether or not to produce a listing of identifiers and attributes.

* Denotes primary control

** In all cases excluding LARGE used with PL/M-86 and PL/M-286. In this case, ROM is the default.

Table 11-1 Compiler Controls (continued)

Control	Default	Abbrev.	Effect
*TITLE	module name in the source code	TT	Places a title on each page of the printed output.
*TYPE *NOTYPE	TYPE	TY NOTY	Specifies whether or not to include type records in the object module.
*XREF *NOXREF	NOXREF	XR NOXR	Enables or disables a cross-reference listing of source program identifiers.

* Denotes primary control

Table 11-1A PL/M-86 Specific Compiler Controls

Control	Default	Abbrev.	Effect
*INTVECTOR *NOINTVECTOR	INTVECTOR	IV NOIV	Enables or disables an interrupt vector.
*MOD86 *MOD186	MOD86	none	Specifies a compiler generated set of microprocessor-specific instructions.

* Denotes primary control

Table 11-1B PL/M-386 Specific Compiler Controls

Control	Default	Abbrev.	Effect
*WORD16 *WORD32	WORD32	W16 W32	Defines the data type terminology.

* Denotes primary control

11.1.1 Input Format Control

LEFTMARGIN

The LEFTMARGIN control specifies the left margin of the source file.

11.1.2 Code Generation and Object File Controls

These controls determine what type of object file is to be produced and in which directory it is to appear. Object file controls include the following:

DEBUG/NODEBUG
INTERFACE
INTVECTOR/NOINTVECTOR
MOD86/MOD186 (for PL/M-86 only)
OBJECT/NOOBJECT
OPTIMIZE
OVERFLOW/NOOVERFLOW
RAM/ROM
SMALL/COMPACT/MEDIUM/LARGE
TYPE/NOTYPE
WORD32/WORD16 (for PL/M-386 only)

11.1.3 Object Module Sections

The output of a PL/M compiler is an object file containing a compiled module. This object module may be linked with other object modules using the appropriate linker or binder. A knowledge of the makeup of an object module is not necessary for PL/M programming, but can aid in understanding the controls for program size and linkage.

PL/M-86

The object module output by the PL/M-86 compiler contains five sections:

- Code Section
- Constant Section (absent in LARGE and in ROM)
- Data Section
- Stack Section
- Memory Section

end PL/M-86

PL/M-286/386

Object modules output by the PL/M-286 and -386 compilers contain three sections:

- Code Section
- Data Section
- Stack Section

These sections can be combined in various ways into memory segments for execution, depending on the size of the program.

end PL/M-286/386

11.1.3.1 Code Section

This section contains the instruction code generated for the source program. If either the LARGE control or the ROM control is used, this section also contains all variables initialized with the DATA attribute, all REAL constants, and all constant lists. (The LARGE control is specific to PL/M-86 and PL/M-286. For upward compatibility, the LARGE control is included in PL/M-386. However, it functions exactly like the COMPACT control.)

In addition, the code section for the main program module contains a main program prologue generated by the compiler. This code precedes the code compiled from the source program, and sets the CPU up for program execution by initializing various registers (and, for the 8086 microprocessor, enabling interrupts).

PL/M-86

11.1.3.2 Constant Section

This section contains all variables initialized with the DATA attribute, as well as all REAL constants and all constant lists. If the LARGE or ROM controls are used, this information is placed in the code section and no constant section is produced.

end PL/M-86

11.1.3.3 Data Section

All variables are allocated space in this section with the exception of parameters, based variables, and variables located with an AT attribute or local to a REENTRANT procedure. (For PL/M-86, this section also excludes variables initialized with the DATA attribute.) If the RAM control is used, this section also contains all variables initialized with the DATA attribute, as well as all REAL constants and all constant lists.

If a nested procedure refers to any parameter of its calling procedure, then all parameters of that calling procedure will be placed in the data section during execution. The compiler reserves enough space during compilation to prepare for this.

11.1.3.4 Stack Section

The stack section is used in executing procedures, as explained in Appendixes F and G. It is also used for any temporary storage used by the program but not explicitly declared in the source module (such as temporary values generated by the compiler).

The exact size of the stack is automatically determined by the compiler except for possible multiple invocations of reentrant procedures. You can override this computation of stack size and explicitly state the stack requirement during the combining process.

NOTE

When using reentrant procedures or interrupt procedures, be sure to allocate a stack section large enough to accommodate all possible storage required by multiple invocations of such procedures. The stack size can be explicitly specified during the module combining process.

The stack space requirement of each procedure is shown in the listing produced by the SYMBOLS or XREF control. This information can be used to compute the additional stack space required for reentrant or interrupt procedures.

PL/M-86

11.1.3.5 Memory Section

This is the area of memory referenced by the built-in PL/M-86 identifier MEMORY. Its maximum allowable size depends on the size control used in compilation (SMALL, COMPACT, MEDIUM, or LARGE) as explained in the following section.

The compiler generates a memory section of length zero, and it is your responsibility to specify the actual (run-time) space required during the module combining process.

end PL/M-86

11.1.4 Segmentation Controls

PL/M-86/286

For PL/M-86 and PL/M-286, these controls specify the memory size requirements of the program containing the module to be compiled. They affect the operation of the compiler in various ways and impose certain constraints on the source module being compiled.

These controls also influence how locations are referenced in the compiled program, which leads to certain programming restrictions for each size control.

Note that for maximum efficiency of the object code, the smallest possible size should be used for any given program. Also note that all modules of a program should be compiled with the same size control. These are primary controls. They have the form:

```
SMALL
COMPACT
MEDIUM
LARGE
```

Extensions to these controls (i.e., the use of subsystems) are discussed in Chapter 13.

end PL/M-86/286

PL/M-386

For PL/M-386, the segmentation controls influence how locations are referenced in the compiled program, which leads to certain programming restrictions for each of the segmentation controls. These are primary controls. They have the following form:

```
SMALL
COMPACT
MEDIUM
LARGE
```

The segmentation controls SMALL and COMPACT determine the maximum allowable size of the segments produced in the object program as well as the grouping of object types (code, data, constants, and stack). These controls affect the operation of the compiler in various ways and impose certain constraints on the source module being compiled.

The MEDIUM control is equivalent to the SMALL control. The LARGE control is equivalent to the COMPACT control except when LARGE is used to indicate a subsystem whose name is unknown at compile time (see Section 13.4).

For maximum efficiency of the object code, the smallest possible size should be used for any given program. Also, all modules of a program should be compiled with the same segmentation control.

The segmentation controls are described later in this chapter (see Section 11.2.22); extensions to these controls (i.e., the use of subsystems) are described in Chapter 13.

end PL/M-386

11.1.5 Listing Selection and Content Controls

These controls determine what types of listings are produced and where they appear. The controls are as follows:

CODE/NOCODE
LIST/NOLIST
PRINT/NOPRINT
SYMBOLS/NOSYMBOLS
XREF/NOXREF

11.1.6 Listing Format Controls

Format controls determine the format of the listing output of the compiler. These controls are as follows:

EJECT
PAGELENGTH
PAGEWIDTH
PAGING/NOPAGING
SUBTITLE
TITLE

11.1.7 Source Inclusion Controls

With these controls, the input source can be changed to a different file. The controls are:

INCLUDE
SAVE/RESTORE

11.1.8 Conditional Compilation Controls

These controls cause selected portions of the source file to be skipped by the compiler if specified conditions are not met. Figure 11-1 shows an example program using conditional compilation and Figure 11-2 shows the same example program using the NOCOND control.

The conditional compilation controls are:

COND/NOCOND
IF/ELSEIF/ELSE/ENDIF
SET/RESET

```
PL/M-86 COMPILER    EXAMPLE                        mm/dd/yy hh:mm:ss   PAGE 1

system-id PL/M-86 Vx.y  COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN cex.obj
COMPILER INVOKED BY:  plm86 cex.src PW(78) SET(DEBUG=3)

   1         EXAMPLE: DO;

   2  1      DECLARE BOOLEAN LITERALLY 'BYTE', TRUE LITERALLY 'OFFH',
                FALSE LITERALLY '0';
   3  1      PRINT$DIAGNOSTICS: PROCEDURE (SWITCHES, TABLES) EXTERNAL;
   4  2      DECLARE (SWITCHES, TABLES) BOOLEAN;
   5  2      END PRINT$DIAGNOSTICS;

   6  2      DISPLAY$PROMPT: PROCEDURE EXTERNAL; END DISPLAY$PROMPT;

   8  2      AWAIT$CR: PROCEDURE EXTERNAL; END AWAIT$CR;

             $IF DEBUG = 1
                CALL PRINT$DIAGNOSTICS (TRUE, FALSE);
             $   RESET (TRAP)
             $ELSEIF DEBUG =2
                CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
             $   RESET (TRAP)
             $ELSEIF DEBUG = 3
  10  1         CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
  11  1         CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
             $   SET (TRAP)
             $ENDIF

             $IF TRAP
  12  1         CALL DISPLAY$PROMPT;
  13  1         CALL AWAIT$CR;
             $ENDIF

  14  1      END EXAMPLE;

MODULE INFORMATION:

     CODE AREA SIZE      = 0017H    23D
     CONSTANT AREA SIZE  = 0000H     0D
     VARIABLE AREA SIZE  = 0000H     0D
     MAXIMUM STACK SIZE  = 0006H     6D
     30 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     4KB MEMORY USED
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

Figure 11-1 Sample Program Using Conditional Compilation (SET Control)

```
PL/M-86 COMPILER EXAMPLE                               mm/dd/yy hh:mm:ss PAGE 1

system-id PL/M-86 Vx.y  COMPILATION OF MODULE EXAMPLE
OBJECT MODULE PLACED IN cex.obj
COMPILER INVOKED BY: plm86 cex.src PW(78) SET(DEBUG=3) NOCOND

   1         EXAMPLE: DO;

   2  1      DECLARE IS LITERALLY 'LITERALLY', BOOLEAN IS 'BYTE', TRUE IS 'OFFH',
                FALSE IS '0';
   3  1      PRINT$DIAGNOSTICS: PROCEDURE (SWITCHES, TABLES) EXTERNAL;
   4  2      DECLARE (SWITCHES, TABLES) BOOLEAN;
   5  2      END PRINT$DIAGNOSTICS;

   6  2      DISPLAY$PROMPT: PROCEDURE EXTERNAL; END DISPLAY$PROMPT;

   8  2      AWAIT$CR: PROCEDURE EXTERNAL; END AWAIT$CR;

             $IF DEBUG = 1
             $ELSEIF DEBUG = 3
  10  1         CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
  11  1         CALL PRINT$DIAGNOSTICS (TRUE, TRUE);
             $   SET (TRAP)
             $ENDIF

             $IF TRAP
  12  1      CALL DISPLAY$PROMPT;
  13  1      CALL AWAIT$CR;
             $ENDIF

  14  1      END EXAMPLE;
```

Figure 11-2 Sample Program Showing the NOCOND Control

11.1.9 Language Compatibility Control

INTERFACE

This control enables PL/M to call procedures written in other languages and vice versa. For PL/M-386, this control also enables the use of external procedures compiled with a PL/M-286 compiler (or other 286-OMF compiler).

11.1.10 Predefined Switches

If one of the switch names (in the following list) appears in an IF or ELSEIF condition and has not been explicitly assigned a value using the SET or RESET control, its default value is its primary control value. (WORD16 and WORD32 are specific to PL/M-386.)

SMALL	MEDIUM	RAM	WORD16
COMPACT	LARGE	ROM	WORD32

If a predefined switch is assigned a value using the SET or RESET control, it functions from that point on like any other switch. A primary control value is not affected by setting or resetting the predefined switch with the same name.

The four model switches are distinct. Even though the primary controls SMALL and MEDIUM have the same control interpretation, specifying the MEDIUM control sets the MEDIUM switch only, and specifying the SMALL control sets the SMALL switch only (similarly for COMPACT and LARGE).

For example, given the following sequence of PL/M-386 control lines:

```
$RAM WORD16 MEDIUM    ; line 1
$IF RAM               ; line 2
 .
$ELSEIF WORD32
 .
$ELSEIF SMALL
 .
$ENDIF
 .
$SET (SMALL, WORD32) ; line x
```

At line 2, the switches RAM and WORD16 are true and their counterparts ROM and WORD32 are false. The switch MEDIUM is true and the switches SMALL, COMPACT, and LARGE are false. Therefore, the IF condition is true and the two ELSEIF conditions are false. After line x, the switches RAM, WORD16, MEDIUM, SMALL, and WORD32 are true; ROM, COMPACT, and LARGE remain false. The setting of SMALL and WORD32 compile time switches (whether set or reset) does not effect the existing segmentation control or any of the other switches.

11.2 Compiler Control Encyclopedia

The following sections present each of the PL/M compiler controls. Note that the segmentation controls are grouped under the SMALL control.

11.2.1 CODE/NOCODE

Form	CODE NOCODE
Default	NOCODE
Type	General

Discussion

The CODE control specifies that listing of the generated object code in pseudo-assembly language format is to begin. This listing is placed at the end of the program listing in the listing file. Note that the CODE control cannot override a NOPRINT control.

The NOCODE control specifies that listing of the generated object code is to be suppressed until the next occurrence, if any, of a CODE control.

11.2.2 COND/NOCOND

These controls determine whether text within an IF element will appear in the listing if it is skipped during compilation.

Form	COND NOCOND
Default	COND
Type	General

Discussion

The COND control specifies that any text that is skipped is to be listed (without statement or level numbers). Note that a COND control cannot override a NOLIST or NOPRINT control, and that a COND control will not be processed if it is within text that is skipped.

The NOCOND control specifies that text within an IF element that is skipped is not to be listed; however, the controls that delimit the skipped text will be listed. This provides an indication that something has been skipped. Note that a NOCOND control will not be processed if it is within text that is skipped.

Figure 11-1 shows an example in which the program was compiled using the COND (by default) and SET controls with the SET switch assignment DEBUG = 3. Figure 11-2 is the same program, but it was compiled using the NOCOND control (see Section 11.1.8). These figures demonstrate the use of conditional compilation. See also the description of SET/RESET.

11.2.3 DEBUG/NODEBUG

Form	DEBUG NODEBUG
Default	NODEBUG
Type	Primary

Discussion

The DEBUG control specifies that the object module is to contain the statement number and relative address of each source program statement, information about each local symbol (including based symbols and procedure parameters), and block information for each procedure. This information may be used later by a debugging tool, such as PSCOPE.

NOTE

OPTIMIZE(0) is the only level of optimization that does not optimize code between program lines. Thus, it is the only one that gives guaranteed results when debugging programs.

11.2.4 EJECT

Form	EJECT
Default	None
Type	General

Discussion

EJECT stops printing on the current page and starts a new page of printed output.

11.2.5 IF/ELSE/ELSEIF/ENDIF

These controls provide conditional compilation capability based on the values of switches.

These controls cannot be used in the invocation of the compiler, and each must be the only control on its control line. There are no default settings or abbreviations for these controls.

An IF control and an ENDIF control delimit an IF element, which can have several different forms. The simplest form of an IF element is:

```
$IF condition
text
$ENDIF
```

Where:

condition is a limited form of a PL/M expression in which the only valid operators are OR, XOR, NOT, AND, <, < =, =, < >, > =, and >, and the only valid operands are switches and whole-number constants with a range of 0 to 255. If the switch does not appear in a SET control, a value of false (0) is assumed (except for predefined switches). Parenthesized subexpressions cannot be used. Within these restrictions, *condition* is evaluated according to the PL/M rules for expression evaluation. Note that *condition* must be followed by an end-of-line.

text is text that will be processed normally by the compiler if the least significant bit of the value of *condition* is a 1, or skipped if the bit is a 0. Note that text can contain any mixture of PL/M source and compiler controls. If the text is skipped, any controls within it are not processed.

The second form of the IF element contains an ELSE element:

```
$IF condition
text 1
$ELSE
text2
$ENDIF
```

In this construction, *text 1* will be processed if the least significant bit of the value of *condition* is a 1, and *text 2* will be skipped. If the bit is a 0, *text 1* will be skipped and *text 2* will be processed.

Only one ELSE control can be used within an IF element.

With the most general form of the IF element, one or more ELSEIF controls can be introduced before the ELSE (if any):

```
$IF condition 1
text 1
$ELSEIF condition 2
text 2
$ELSEIF condition 3
text 3
 .
 .
 .
$ELSEIF condition n
text n
$ELSE
text n + 1
$ENDIF
```

where any of the ELSEIF elements can be omitted, as can the ELSE element.

The conditions are tested in sequence. As soon as one of them yields a value with a 1 as its least significant bit, the associated text is processed. All other text in the IF element is skipped. If none of the conditions yields a least significant bit of 1, the text in the ELSE element (if any) is processed and all other text in the IF element is skipped.

Parentheses cannot be used on a conditional control line. For example:

```
$IF A+(B+C)
```

is illegal.

11.2.6 INCLUDE

Form INCLUDE (*pathname*)

Default None

Type General

Discussion

An INCLUDE control must be the right-most control in a control line or in the invocation command.

The INCLUDE control causes the specified file to be included during compilation. Input continues from this file until an end-of-file is detected, and then processing resumes in the file containing the INCLUDE control.

An included file may also contain INCLUDE controls. Note that such nesting of included files cannot exceed the depth as given in Appendix B.

11.2.7 INTERFACE

PL/M-86/286

In PL/M-86 and PL/M-286, INTERFACE is a primary control that enhances the compatibility of PL/M with other programming languages. The INTERFACE control enables PL/M programs to call procedures written in other languages. Additionally, with the INTERFACE control, procedures written in PL/M can be called by procedures written in other languages. The calling conventions for procedures written in Pascal, Fortran, and PL/M are identical; therefore, the only language that needs a special designation for its calling convention is C.

This control has the following form: (Note that INTERFACE cannot be part of an invocation command.)

Form INTERFACE (*lang* = *name1* [. . .])

Default None

Type Primary

Where:

lang	is the name of the language that requires a different calling convention for procedures; in this case, the language is C. Specifying INTERFACE for a language other than C has no effect; the standard calling convention for PL/M will be used.
name	represents the procedure or data structure that will be called or referenced from the PL/M program. The compiler generates code that enables procedures to be called according to the calling convention of the specified language.

end
PL/M-86/286

PL/M-386

For PL/M-386, INTERFACE is a primary control that enables PL/M-386 programs to call or be called by procedures written in C-386 and C-286, or procedures compiled with an 80286 translator (specifically PL/M-286, Fortran-286, Pascal-286, and ASM286).

This control has the following form: (Note that INTERFACE cannot be part of an invocation command.)

Form	INTERFACE (*lang* [/*machine* [/*model* [/*ramrom*]]] [= *id* [,*id*]. . .])
Default	None
Type	Primary

Where:

lang	is C, the name of the language that requires a different calling convention for procedures.
machine	is 386 when calling C-386, and 286 when calling languages compiled using an 80286 translator. Only references to 286 *id*s from 386 modules are supported; 386 *id*s cannot be referenced from 286 modules. Therefore, if *machine* is 286 then all the identifiers in the *id* list must be declared EXTERNAL. If MACHINE is 286 and an *id* is PUBLIC, it is an error.

model	is SMALL, COMPACT, MEDIUM, or LARGE and defines the model of segmentation for the specified *ids*. *model* determines whether 286 POINTER variables are offset-only or selector-offset, as defined by the PL/M-286 models of segmentation (see Chapter 13, Table 13-1). If *machine* is 286, *model* defaults to LARGE. *model* should be specified as the same model of segmentation used to compile the 80286 code being referenced. If *machine* is 386, *model* is ignored.
ramrom	is RAM or ROM and defines the placement of constant variables in either the code or data segment. When used with the model SMALL, *ramrom* also defines whether POINTER variables are offset-only or selector-offset. The default is RAM unless *model* is LARGE, in which case the default is ROM. *ramrom* is ignored if *machine* is 386.
id	specifies the procedures and variables that are implemented using the specified language interface convention.

Discussion

When the INTERFACE control is used to call procedures compiled with an 80286 translator, the program switches from using 32-bit stack offsets to 16-bit offsets. Therefore, the stack pointer for the called procedure must point within the lowest 64K of the stack segment, or else a gate must be used to switch to such a stack segment. Parameters must fit within this boundary as well.

The calling conventions for the 80386 languages (except C-386) are identical to PL/M-386 and therefore do not require the use of the INTERFACE control. Because the calling conventions differ for C-386, INTERFACE must be used to call or to be called from C-386 procedures. The C interface convention differs from the PL/M calling convention in the following ways:

- Parameters are evaluated and pushed onto the stack in the reversed manner.
- 8- and 16-bit parameters are zero extended or sign extended to 32 bits.
- Real parameters for C are always 64-bit double floating-point numbers and are passed on the 80386 stack.
- The caller clears the parameters off the stack after return and the callee does not pop parameters off the stack.

The following example demonstrates the use of the INTERFACE control to call a PL/M-286 procedure:

```
$WORD16
$INTERFACE(PLM/286/SMALL/ROM=EXAMP)

EXAMP:PROCEDURE(A,B)EXTERNAL;
   DECLARE A WORD, B POINTER;
END EXAMP;

DECLARE X WORD, Y POINTER;

CALL EXAMP(X,Y);
```

In the preceding example, the INTERFACE control specifies the procedure EXAMP to be defined as an 80286-compatible. The actual parameters X and Y will be automatically converted to a 16-bit WORD and a 32-bit 80286 POINTER, respectively.

The next example demonstrates calling a C-386 procedure:

```
$INTERFACE(C/386=foo)
foo: procedure(a,b,c);
   declare (a,b,c) integer; end;
declare (i,j,k) integer;
call foo (i,j,k);
```

In this example, INTERFACE specifies the procedure "foo" to be defined in an 80386-compatible segment using the C calling conventions. The compiler automatically pushes the actual parameters in the correct order for the C procedure.

Because Intel's C language requires that all 8- and 16-bit integer parameters be passed as 32-bit values, formal parameters of C-386 procedures should be declared as 32-bit PL/M types: WORD or INTEGER. In this way, the PL/M-386 compiler will convert all actual parameters to the type expected by the C-386 code or passed from the C-386 code.

80286 variables and formal parameters of 80286 procedures should be declared the same as in the 80286 code. The PL/M-386 compiler is able to interpret the terms in an 80286 context and perform the following mapping:

Term Used	Maps to Data Type
BYTE	8-bit, unsigned
HWORD	8-bit, unsigned
WORD	16-bit, unsigned
DWORD	32-bit, unsigned
QWORD	32-bit, unsigned
CHARINT	8-bit (interpretation dependent on 80286 code)
SHORTINT	8-bit (interpretation dependent on 80286 code)
INTEGER	16-bit, signed integer
LONGINT	32-bit (interpretation dependent on 80286 code)
REAL	32-bit, real
SELECTOR	16-bit, selector
POINTER	see the following paragraphs
OFFSET	16-bit, unsigned

An 80286-style long POINTER (32 bits composed of a 16-bit selector and a 16-bit offset), is a data type not defined in the PL/M-386 language. The PL/M-386 compiler converts 48-bit 80386-style long POINTERs to 80286 POINTERs by truncating the offset portion to 16 bits. In SMALL RAM, a POINTER is the same as an OFFSET, and is treated as such by the compiler.

Note that this mapping is independent of WORD16/WORD32 (defined in Tables 9-3, 10-1, and 11-3). This means that there is a third mapping of scalar terms to scalar data types.

end
PL/M-386

PL/M-86

11.2.8 INTVECTOR/NOINTVECTOR

Form	INTVECTOR NOINTVECTOR
Default	INTVECTOR
Type	Primary

Discussion

Under the INTVECTOR control, the compiler creates an interrupt vector consisting of a 4-byte entry for each interrupt procedure in the module. For interrupt *n*, the interrupt vector entry is located at absolute location 4**n*. See Chapter 8 and Appendix G for further discussion about procedures and interrupt processing.

Alternatively, it may be desirable to create the interrupt vector independently, using either PL/M-86 or assembly language. In this case, the NOINTVECTOR control is used and the compiler does not generate an interrupt vector. The implications of this are discussed in Appendix G.

end
PL/M-86

11.2.9 LEFTMARGIN

This is the only control for specifying the format of the source input.

Form	LEFTMARGIN(*n*)
Default	LEFTMARGIN(1)
Type	General

Discussion

All characters to the left of position *n* on subsequent input lines are not processed by the compiler (but do appear on the listing). The first character on a line is in column 1.

The new setting of the left margin takes effect on the next input line. It remains in effect for all input from this source file and any related INCLUDE files until it is reset by another LEFTMARGIN control.

Note that a control line is one that contains a dollar sign in the column specified by the most recent LEFTMARGIN control.

11.2.10 LIST/NOLIST

Form	LIST NOLIST
Default	LIST
Type	General

Discussion

The LIST control specifies that listing of the source program is to resume with the next source line read. Note that the LIST control cannot override a NOPRINT control. If NOPRINT is in effect, no listing is produced.

The NOLIST control specifies that listing of the source program is to be suppressed until the next occurrence, if any, of a LIST control.

When LIST is in effect, all input lines (from the source file or from an INCLUDE file), including control lines, are listed, provided there is not a NOPRINT control in effect. When NOLIST is in effect, only source lines associated with error messages are listed.

PL/M-86

11.2.11 MOD86/MOD186

Form	MOD86 MOD186
Default	MOD86
Type	Primary

Discussion

The MOD86 control specifies that the object module will contain instructions for execution on the 8086 microprocessor.

The MOD186 control enables the compiler to generate an extended set of instructions in the object module for use on the 80186 microprocessor.

This control must be specified when using the block I/O procedures described in Chapter 9 (see Section 9.4.8).

11.2.12 OBJECT/NOOBJECT

Form	OBJECT(*pathname*) NOOBJECT
Default	OBJECT(*sourcefilename*.OBJ)
Type	Primary

Discussion

The OBJECT control specifies that an object module is to be created during compilation. The *pathname* is a standard host operating system *pathname* that specifies the file to receive the object module. If the control is absent or if an OBJECT control appears without a *pathname*, the object module is directed to a file that has the same name as the source input file, but with the extension .OBJ.

The NOOBJECT control specifies that no object module is to be produced.

11.2.13 OPTIMIZE

This control governs the level of optimization to be performed in generating object code. The *n* shown in the syntax represents the various levels of optimization, ranging from zero (the lowest) to three (the highest). Figures 11-3 to 11-6 illustrate the different levels of optimization. The same program was run for each level, but, the source file was printed only for OPTIMIZE (0).

Form OPTIMIZE(*n*)

Where: *n* = (0, 1, 2, 3)

Default OPTIMIZE (1)

Type Primary

Discussion

OPTIMIZE (0). OPTIMIZE(0) specifies only folding of constant expressions.

Folding means recognizing, during compilation, operations that are superfluous or combinable, and removing or combining them so as to save memory space or execution time. Examples include addition with a zero operand, multiplication by one, and logical expressions with true or false constants.

OPTIMIZE(0) is the only level of optimization that is guaranteed to not optimize code between lines.

Figure 11-3 illustrates the OPTIMIZE (0) level of optimization.

```
PL/M-86 COMPILER   EXAMPLES_OF_OPTIMIZATIONS       mm/dd/yy hh:mm:ss  PAGE 1
system-id PL/M-86 Vx.y COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN example.obj
COMPILER INVOKED BY:  plm86 example.src PW(78) COMPACT CODE OPTIMIZE(0)

   1          EXAMPLES_OF_OPTIMIZATIONS: DO;
   2  1          DECLARE (A,B,C) WORD,
                          D(100) WORD,
                         (PTR_1, PTR_2) POINTER,
                          ABASED BASED PTR_1 (10) WORD;
   3  1          DO WHILE D(A+B) < D(A+B+1);
   4  2             IF PTR_P1 < PTR_2 THEN DO;
   6  3                A = A * 2;
   7  3                ABASED(A) = ABASED(B);
   8  3                ABASED(B) = ABASED(C);
   9  3                END;
  10  2              ELSE A = A + 1;
  11  2            END;
  12  1       END EXAMPLES_OF_OPTIMIZATIONS;
                                                        ; STATEMENT # 3
           0000  8BEC          MOV    BP,SP
           0002  FB            STI
                    @1:
           0003  8B1E0000      MOV    BX,A
           0007  031E0200      ADD    BX,B
           000B  D1E3          SHL    BX,1
           000D  8B360000      MOV    SI,A
           0011  03360200      ADD    SI,B
           0015  46            INC    SI
           0016  D1E6          SHL    SI,1
           0018  8B870600      MOV    AX,D[BX]
           001C  3B840600      CMP    AX,D[SI]
           0020  7203          JB     $+5H
           0022  E97C00        JMP    @2
                                                        ; STATEMENT # 4
           0025  C406CE00      LES    AX,PTR_1
           0029  B104          MOV    CL,4H
           002B  8CC6          MOV    SI,ES
           002D  8BD0          MOV    DX,AX
           002F  D3EA          SHR    DX,CL
           0031  03F2          ADD    SI,DX
           0033  81E00F00      AND    AX,0FH
           0037  C416D200      LES    DX,PTR_2
           003B  B104          MOV    CL,4H
```

Figure 11-3 Sample Program Showing the OPTIMIZE(0) Control

```
                    ASSEMBLY LISTING OF OBJECT CODE

            003D  8CC7                 MOV   DI,ES
            003F  8BDA                 MOV   BX,DX
            0041  D3EB                 SHR   BX,CL
            0043  03FB                 ADD   DI,BX
            0045  81E20F00             AND   DX,0FH
            0049  3BF7                 CMP   SI,DI
            004B  7502                 JNZ   $+4H
            004D  3BC2                 CMP   AX,DX
            004F  7403                 JZ    $+5H
            0051  E94100               JMP   @3
                                                   ; STATEMENT # 6
            0054  8B060000             MOV   AX,A
            0058  D1E0                 SHL   AX,1
            005A  89060000             MOV   A,AX
                                                   ; STATEMENT # 7
            005E  8B1E0200             MOV   BX,B
            0062  D1E3                 SHL   BX,1
            0064  8B360000             MOV   SI,A
            0068  D1E6                 SHL   SI,1
            006A  C43ECE00             LES   DI,PTR_1
            006E  268B01               MOV   AX,ES:[BX].ABASED[DI]
            0071  C41ECE00             LES   BX,PTR_1
            0075  268900               MOV   ES:[BX].ABASED[SI],AX
                                                   ; STATEMENT # 8
            0078  8B1E0400             MOV   BX,C
            007C  D1E3                 SHL   BX,1
            007E  8B360200             MOV   SI,B
            0082  D1E6                 SHL   SI,1
            0084  C43ECE00             LES   DI,PTR_1
            0088  268B01               MOV   AX,ES:[BX].ABASED[DI]
            008B  C41ECE00             LES   BX,PTR_1
            008F  268900               MOV   ES:[BX].ABASED[SI],AX
            0092  E90900               JMP   @4
                                                   ; STATEMENT # 10
                             @3:
            0095  8B060000             MOV   AX,A
            0099  40                   INC   AX
            009A  89060000             MOV   A,AX
```

Figure 11-3 Sample Program Showing the OPTIMIZE(0) Control (continued)

```
PL/M-86 COMPILER   EXAMPLES_OF_OPTIMIZATIONS     mm/dd/yy hh:mm:ss  PAGE 3
                      ASSEMBLY LISTING OF OBJECT CODE
                                                  ; STATEMENT # 11
                           @4:
          009E  E962FF          JMP     @1
                           @2:
                                                  ; STATEMENT # 12
MODULE INFORMATION:

     CODE AREA SIZE      = 00A1H   161D
     CONSTANT AREA SIZE = 0000H   0D
     VARIABLE AREA SIZE = 00D6H   214D
     MAXIMUM STACK SIZE = 0000H   0D
     16 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS
DICTIONARY SUMMARY:
     4KB MEMORY USED
     0KB DISK SPACE USED
END OF PL/M-86 COMPILATION
```

Figure 11-3 Sample Program Showing the OPTIMIZE(0) Control (continued)

OPTIMIZE(1). OPTIMIZE(1) specifies strength reduction, elimination of common subexpressions, short-circuit evaluation of some Boolean expressions, as well as the optimizations of level (0).

Strength reduction means substituting quick operations in place of longer operations (e.g., shifting by 1 instead of multiplying by 2). This instruction requires less space and executes faster. The occurrence of identical subexpressions may also generate left shift instructions.

Elimination of common subexpressions means that if an expression reappears in the same block, its value is re-used rather than recomputed. The compiler also recognizes commutative forms of subexpressions (e.g., A + B and B + A are seen as the same). Intermediate results during expression evaluation are saved in either registers or on the stack for later use. For example:

```
A = B + C * D / 3;
C = E + D * C / 3;
```

The value of C*D/3 will not be recomputed for the second statement.

Optimizing the evaluation of Boolean expressions uses the fact that in certain cases some of the terms are not needed to determine the value of the expression. For example, in the expression:

```
( A > B AND I > J )
```

if the first term (A > B) is false, the entire expression is false, and it is not necessary to evaluate the second term. The use of PL/M built-in procedures does not change this optimization. However, if a user-function or an embedded assignment is part of the expression, this short evaluation is not done. For example:

```
(A > B AND ( UFUN (A) > J ) )
```

is evaluated in full.

Figure 11-4 illustrates the OPTIMIZE (1) level of optimization.

```
PL/M-86 COMPILER   EXAMPLES_OF_OPTIMIZATIONS     mm/dd/yy hh:mm:ss  PAGE 1

system-id PL/M-86 Vx.y  COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN example.obj
COMPILER INVOKED BY: plm86 example.src PW(78) COMPACT CODE OPTIMIZE(1)
                     NOLIST

                                             ; STATEMENT # 3
          0000  8BEC             MOV    BP,SP
          0002  FB               STI
                          @1:
          0003  8B1E0000         MOV    BX,A
          0007  8B360200         MOV    SI,B
          000B  03DE             ADD    BX,SI
          000D  53               PUSH   BX ; 1
          000E  D1E3             SHL    BX,1
          0010  5F               POP    DI ; 1
          0011  47               INC    DI
          0012  D1E7             SHL    DI,1
          0014  8B870600         MOV    AX,D[BX]
          0018  3B850600         CMP    AX,D[DI]
          001C  7203             JB     $+5H
          001E  E96D00           JMP    @2
```

Figure 11-4 Sample Program Showing the OPTIMIZE(1) Control

```
                ASSEMBLY LISTING OF OBJECT CODE
                                                ; STATEMENT # 4
                0021  C406CE00        LES    BX,PTR_1
                0025  B104            MOV    CL,4H
                0027  8CC7            MOV    DI,ES
                0029  8DB0            MOV    DX,AX
                002B  D3EA            SHR    DX,CL
                002D  03FA            ADD    DI,DX
                002F  81E00F00        AND    AX,0FH
                0033  C416D200        LES    DX,PTR_2
                0037  8CC3            MOV    BX,ES
                0039  8BF2            MOV    SI,DX
                003B  D3EE            SHR    SI,CL
                003D  03DE            ADD    BX,SI
                003F  81E20F00        AND    DX,0FH
                0043  3BFB            CMP    DI,BX
                0045  7502            JNZ    $+4H
                0047  3BC2            CMP    AX,DX
                0049  7203            JB     $+5H
                004B  E93900          JMP    @3
                                                ; STATEMENT # 6
                004E  8B1E0000        MOV    BX,A
                0052  D1E3            SHL    BX,1
                0054  891E0000        MOV    A,BX
                                               ; STATEMENT # 7
                0058  8B360200        MOV    SI,B
                005C  D1E6            SHL    SI,1
                005E  D1E3            SHL    BX,1
                0060  53              PUSH   BX ; 1
                0061  C1ECE00         LES    BX,PTR_1
                0065  268B00          MOV    AX,ES:[BX].ABASED[SI]
                0068  5E              POP    SI ; 1
                0069  268900          MOV    ES:[BX].ABASED[SI],AX
                                                ; STATEMENT # 8
                006C  8B1E0400        MOV    BX,C
                0070  D1E3            SHL    BX,1
                0072  8B360200        MOV    SI,B
                0076  D1E6            SHL    SI,1
                0078  C43ECE00        LES    DI,PTR_1
                007C  268B01          MOV    AX,ES:[BX].ABASED[DI]
                007F  8BDE            MOV    BX,SI
                0081  268901          MOV    ES:[BX].ABASED[DI],AX
                0084  E90400          JMP    @4
```

Figure 11-4 Sample Program Showing the OPTIMIZE(1) Control (continued)

```
PL/M-86 COMPILER    EXAMPLES_OF_OPTIMIZATIONS        mm/dd/yy hh:mm:ss   PAGE 3
           ASSEMBLY LISTING OF OBJECT CODE
                                                    ; STATEMENT # 10
                         @3:
           0087  FF060000         INC     A
                                                    ; STATEMENT # 11
                         @4:
           008B  E975FF           JMP     @1
                         @2:
                                                    ; STATEMENT # 12

MODULE INFORMATION:

     CODE AREA SIZE     = 008EH    142D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 00D6H    214D
     MAXIMUM STACK SIZE = 0002H      2D
     16 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     4KB MEMORY USED
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

Figure 11-4 Sample Program Showing the OPTIMIZE(1) Control (continued)

OPTIMIZE(2). OPTIMIZE(2) includes OPTIMIZE(0) and OPTIMIZE(1), plus the following:

- Machine code optimizations (e.g., short jumps, moves)
- Elimination of superfluous branches
- Re-use of duplicate code
- Removal of unreachable code and reversal of branch conditions

Optimizing machine code means saving space by using shorter forms for identical machine instructions. This is possible because the microprocessor has multiple forms for some of its instructions. For example:

```
MOV RESLT1, AX;
          /* move accumulator value to location RESLT1 */
```

can be generated using three or four bytes for PL/M-86 and PL/M-286, and using five or six bytes for PL/M-386. The former choice saves a byte of storage for the program. Similarly, jumps that the compiler can recognize as within the same segment or closer (within 127 bytes) permit the use of fewer byte instructions.

Elimination of superfluous branches means optimizing consecutive or multiple branches into a single branch. For example:

```
   JZ     LAB1;                      /* Jump on zero to LAB1 */
   JMP    LAB2;               /* unconditional jump to LAB2 */
  LAB1:   .....
           ...
           ...
  LAB2:   .....
```

will be transformed into:

```
   JNZ   LAB2;                   /* Jump on non-zero to LAB2 */
  LAB1:  .....
          ...
          ...
  LAB2:  .....
```

Similarly, multiple branches like the following are eliminated:

```
  LAB0:JMP  LAB1
             ...
             ...
  LAB1:JMP  LAB2
             ...
  LAB2:.....
```

and transformed into:

```
  LAB0:JMP  LAB2
             ...
             ...
  LAB1:JMP  LAB2
             ...
  LAB2:.....
```

Reuse of duplicate code can refer to identical code at the end of two converging paths. In such a case, the code is inserted in only one path, and a jump to that path is inserted in the other path. For example:

```
  DECLARE A BYTE, SPOT POINTER;
  DECLARE S BASED SPOT STRUCTURE (B BYTE, C BYTE);
  IF A = 1 THEN
  S.C = INPUT (0F7H) AND 07FH;
  ELSE
  S.C = INPUT (0F9H) AND 07FH;
```

```
                 Before                              After

           CMP   A, 1H                       CMP   A, 1H
           JZ    & + 5H                      JMP   @1
           JMP   @1
           IN    0F7H                        IN    0F7H
           AND   AL, 7FH                     JMP   @2
           MOV   BX, SPOT
           MOV   S [BX+1H], AL
           JMP   @2
     @1:   IN    0F9H                    @1: IN    0F9H
           AND   AL, 7FH                 @2: AND   AL, 7FH
           MOV   BX, SPOT                    MOV   BX, SPOT
           MOV   S [BX+1H], AL               MOV   S [BX+1H],AL
     @2:
```

Reuse of duplicate code can also refer to machine instructions, immediately preceding a loop, that are identical to those ending the loop. A branch can be generated to reuse the code generated at the beginning of the loop. For example:

```
                  Before                          After

             ADD   AX, BX                 LAB0: ADD AX, BX
             MOV   ANS, AX                      MOV ANS, AX
   LAB0:     MOV   AL, DUM1                     MOV AL, DUM1
             CMP   AL, DUM2                     CMP AL, DUM2
             JNZ   LAB1                         JNZ LAB1
             ...                                ...
             ...                                ...
             ADD   AX, BX                       JMP LAB0
             MOV   ANS, AX                LAB1: ...
             JMP   LAB0
   LAB1:     ....
```

This is safe so long as LAB0 is not the target of a jump instruction. The compiler normally handles a whole procedure at a time, and is aware of such a condition. This optimization cannot be safely applied to labels in the outer level of the main program module. This optimization will not change the program and will save code space.

Second level optimization removes unreachable code, takes a second look at the generated object code, and finds areas that can never be reached due to the control structures created earlier.

For example, if the following code were generated before optimization:

```
        MOV     AX, A
        RCR     AL, 1
        JB      @1
        JMP     @2

@1:     MOV     AX, 0FFFFH
        OUTW    0F6H
        JMP     @2
        MOV     AX, B
        ADD     A, AX
        JMP     @3
@2:     ....
        ....
        ....
        ....
@3:     .....
        .....
```

then the removal of unreachable code would produce:

```
        MOV     AX, A
        RCR     AL, 1
        JB      @1
        JMP     @2

@1:     MOV     AX,0FFFFH
        OUTW    0F6H
        JMP     @2
@2:     ...
        ...
@3:     .....
```

This can be further optimized by reversing the branch condition in the third instruction and removing the unnecessary JMP @2:

```
        MOV     AX, A
        RCR     AL, 1
        JNB     @2

@1:     MOV     AX, 0FFFFH
        OUTW    0F6H
@2:     ...
        ...
@3:     .....
```

Figure 11-5 illustrates the OPTIMIZE (2) level of optimization.

```
PL/M-86 COMPILER   EXAMPLES_OF_OPTIMIZATIONS     mm/dd/yy hh:mm:ss  PAGE 1
system-id PL/M-86 Vx.y COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN example.obj
COMPILER INVOKED BY: plm86 example.src PW(78) COMPACT CODE OPTIMIZE(2)
                     NOLIST

                                                  ; STATEMENT # 3
           0000  8BEC             MOV    BP,SP
           0002  FB               STI
                         @1:
           0003  8B1E0000         MOV    BX,A
           0007  A10200           MOV    AX,B
           000A  03D8             ADD    BX,AX
           000C  53               PUSH   BX ; 1
           000D  D1E3             SHL    BX,1
           000F  5E               POP    SI ; 1
           0010  46               INC    SI
           0011  D1E6             SHL    SI,1
           0013  8B8F0600         MOV    CX,D[BX]
           0017  3B8C0600         CMP    CX,D[SI]
           001B  7366             JNB    @2
                                                  ; STATEMENT # 4
           001D  C416CE00         LES    BX,PTR_1
           0021  B104             MOV    CL,4H
           0023  8CC6             MOV    SI,ES
           0025  8BFA             MOV    DI,DX
           0027  D3EF             SHR    DI,CL
           0029  03F7             ADD    SI,DI
           002B  81E20F00         AND    DX,0FH
           002F  C41ED200         LES    BX,PTR_2
           0033  8CC7             MOV    DI,ES
           0035  8BC3             MOV    AX,BX
           0037  D3E8             SHR    AX,CL
           0039  03F8             ADD    DI,AX
           003B  81E30F00         AND    BX,0FH
           003F  3BF7             CMP    SI,DI
           0041  7502             JNZ    $+4H
           0043  3BD3             CMP    DX,BX
           0045  7336             JNB    @3
                                                  ; STATEMENT # 6
           0047  A10000           MOV    AX,A
           004A  D1E0             SHL    AX,1
           004C  A30000           MOV    A,AX
                                                  ; STATEMENT # 7
           004F  8B1E0200         MOV    BX,B
           0053  D1E3             SHL    BX,1
```

Figure 11-5 Sample Program Showing the OPTIMIZE(2) Control

```
PL/M-86 COMPILER   EXAMPLES_OF_OPTIMIZATIONS        mm/dd/yy hh:mm:ss  PAGE 2
          ASSEMBLY LISTING OF OBJECT CODE

          0055  D1E0           SHL    AX,1
          0057  C436CE00       LES    SI,PTR_1
          005B  268B08         MOV    CX,ES:[BX].ABASED[SI]
          005E  8BD8           MOV    BX,AX
          0060  26890B         MOV    ES:[BX].ABASED[SI],CX
                                                  ; STATEMENT # 8
          0063  8B1E0400       MOV    BX,I
          0067  D1E3           SHL    BX,1
          0069  8B360200       MOV    SI,B
          006D  D1E6           SHL    SI,1
          006F  C43ECE00       LES    DI,PTR_1
          0073  268B01         MOV    AX,ES:[BX].ABASED[DI]
          0076  8BDE           MOV    BX,SI
          0078  268901         MOV    ES:[BX].ABASED[DI],AX
          007B  EB86           JMP    @1
                                                  ; STATEMENT # 10
                         @3:
          007D  FF060000       INC    A
                                                  ; STATEMENT # 11
          0081  EB80           JMP    @1
                         @2:
                                                  ; STATEMENT # 12

MODULE INFORMATION:

     CODE AREA SIZE      = 0083H  131D
     CONSTANT AREA SIZE  = 0000H    0D
     VARIABLE AREA SIZE  = 00D6H  214D
     MAXIMUM STACK SIZE  = 0002H    2D
     16 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:


     4KB MEMORY USED
     0KB DISK SPACE USED


END OF PL/M-86 COMPILATION
```

Figure 11-5 Sample Program Showing the OPTIMIZE(2) Control (continued)

OPTIMIZE(3). OPTIMIZE(3) includes all of the preceding optimizations. It also optimizes indeterminate storage operations (e.g., those using based variables or variables declared with the AT attribute).

NOTE

The assumption validating this new optimization is that based variables (or AT variables) do not overlay other user-declared variables.

On this optimization level, all Boolean expressions are short-circuited except those containing embedded assignments. (For a description of how this optimization occurs, see OPTIMIZE(1).)

The benefits of this optimization level include more efficient use of code space only if needed values are not overlaid.

Caution in variable-declaration and usage is essential. For example, the sequence:

```
DECLARE (I, J) WORD;
DECLARE THETA (19) AT (@I);
DECLARE A BASED J (10);
STRUCTURE (F1 BYTE, F2 WORD);
..
J=.I;
....
..
A(I).F1 = 7;
A(I).F2 = 99;
THETA(I) = 31;
..
..
```

violates this caution because it causes the values being used as pointers and subscripts to be overlaid. The compiler normally takes steps to avoid the difficulties implied here. But, in OPTIMIZE(3), these steps are omitted due to the implicit requirement that such situations must not be present at this level of optimization.

Figure 11-6 illustrates the OPTIMIZE (3) level of optimization.

```
PL/M-86 COMPILER    EXAMPLES_OF_OPTIMIZATIONS    mm/dd/yy hh:mm:ss  PAGE 1

system-id PL/M-86 Vx.y COMPILATION OF MODULE EXAMPLES_OF_OPTIMIZATIONS
OBJECT MODULE PLACED IN example.obj
COMPILER INVOKED BY: plm86 example.src PW(78) COMPACT CODE OPTIMIZE(3)
                NOLIST
                                                  ; STATEMENT # 3
        0000  8BEC            MOV    BP,SP
        0002  FB              STI
                    @1:
        0003  8B1E0000        MOV    BX,A
        0007  A10200          MOV    AX,B
        000A  03D8            ADD    BX,AX
        000C  53              PUSH   BX ; 1
        000D  D1E3            SHL    BX,1
        000F  5E              POP    SI ; 1
        0010  46              INC    SI
        0011  D1E6            SHL    SI,1
        0013  8B8F0600        MOV    CX,D[BX]
        0017  3B8C0600        CMP    CX,D[SI]
        001B  733B            JNB    @2
                                                  ; STATEMENT # 4
        001D  C41ECE00        LES    BX,PTR_1
        0021  8CC6            MOV    SI,ES
        0023  3B36D400        CMP    SI,PTR_2+2H
        0027  7504            JNZ    $+6H
        0029  3B1ED200        CMP    BX,PTR_2
        002D  7523            JNZ    @3
                                                  ; STATEMENT # 6
        002F  8B3E0000        MOV    DI,A
        0033  D1E7            SHL    DI,1
        0035  893E0000        MOV    A,DI
                                                  ; STATEMENT # 7
        0039  D1E0            SHL    AX,1
        003B  D1E7            SHL    DI,1
        003D  96              XCHG   AX,SI
        003E  268B08          MOV    CX,ES:[BX].ABASED[SI]
        0041  268909          MOV    ES:[BX].ABASED[DI],CX
                                                  ; STATEMENT # 8
        0044  8B3E0400        MOV    DI,C
        0048  D1E7            SHL    DI,1
        004A  268B09          MOV    CX,ES:[BX].ABASED[DI]
        004D  268908          MOV    ES:[BX].ABASED[SI],CX
        0050  EBB1            JMP    @1
```

Figure 11-6 Sample Program Showing the OPTIMIZE(3) Control

```
PL/M-86 COMPILER   EXAMPLES_OF_OPTIMIZATIONS          mm/dd/yy hh:mm:ss  PAGE 2
                      ASSEMBLY LISTING OF OBJECT CODE
                                                      ; STATEMENT # 10
                             @3:
          0052   FF060000          INC     A
                                                      ; STATEMENT # 11
          0056   EBAB              JMP     @1
                             @2:
                                                      ; STATEMENT # 12

MODULE INFORMATION:

     CODE AREA SIZE      = 0058H    88D
     CONSTANT AREA SIZE  = 0000H     0D
     VARIABLE AREA SIZE  = 00D6H   214D
     MAXIMUM STACK SIZE  = 0002H     2D
     16 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     4KB MEMORY USED
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

Figure 11-6 Sample Program Showing the OPTIMIZE(3) Control (continued)

11.2.14 OVERFLOW/NOOVERFLOW

Form	OVERFLOW NOOVERFLOW
Default	NOOVERFLOW
Type	General

Discussion

These controls specify whether overflow is to be detected in performing signed arithmetic. If the NOOVERFLOW control is specified, no overflow detection is implemented in the compiled module and the results of overflow in signed arithmetic are undefined. If the OVERFLOW control is specified, overflow in signed arithmetic results in a nonmaskable interrupt 4, and it is the programmer's responsibility to provide an interrupt procedure to handle the interrupt. Failure to provide such a procedure may result in unpredictable program behavior when overflow occurs.

If this control is nested within a program statement, overflow detection will begin when the next complete statement is evaluated.

Note that the use of the OVERFLOW control results in some expansion of the object code.

Specific to the 80386 microprocessor, in-line checking code is inserted for detecting machine overflow (32-bit arithmetic overflow) on signed expressions, and value overflow on assignments to SHORTINT or CHARINT variables.

To save code space and execution time, avoid using SHORTINT and CHARINT when compiling with the OVERFLOW control.

11.2.15 PAGELENGTH

Form	PAGELENGTH(*n*)
Default	PAGELENGTH(60)
Type	Primary

Discussion

Pagelength is a non-zero, unsigned number specifying the maximum number of lines to be printed per page of listing output. This number is taken to include the page headings appearing on the page.

The minimum value for length is 5; the maximum value is 255.

11.2.16 PAGEWIDTH

Form	PAGEWIDTH(*n*)
Default	PAGEWIDTH(120)
Type	Primary

Discussion

Pagewidth is a non-zero, unsigned number specifying the maximum line width, in characters, to be used for listing output. The minimum value for width is 60; the maximum value is 132.

11.2.17 PAGING/NOPAGING

Form	PAGING NOPAGING
Default	PAGING
Type	Primary

Discussion

The PAGING control specifies that the listed output is to be formatted onto pages. Each page carries a heading identifying the compiler and a page number, and possibly a user specified title.

The NOPAGING control specifies that page ejecting, page heading, and page numbering are not to be performed. Thus, the listing appears on one long page as would be suitable for a slow serial output device. If NOPAGING is specified, a page eject is not generated if an EJECT control is encountered.

11.2.18 PRINT/NOPRINT

Form	PRINT(*pathname*) NOPRINT
Default	PRINT(*sourcefilename*.LST)
Type	Primary

Discussion

The PRINT control specifies that printed output is to be produced. *Pathname* is a standard host operating system *pathname* that specifies the file to receive the printed output. Any output-type device, including a disk file, can also be given. If the control is absent, or if a PRINT control appears without a *pathname*, printed output is sent to a file that has the same name as the source input file but with the extension .LST.

The NOPRINT control specifies that no printed output is to be produced, even if implied by other listing controls such as LIST and CODE.

11.2.19 RAM/ROM

Form	RAM ROM
Default	RAM (Except in the LARGE model for PL/M-86 and PL/M-286, in which case ROM is the default.)
Type	Primary

Discussion

Specific to PL/M-86, the RAM setting directs the object-module placement of all constants, both user-defined and compiler-generated. The default setting places the CONSTANT section within the DATA segment in all segmentation models (sizes) except LARGE. In the LARGE model, with the ROM setting, constants are placed in the CODE segment.

For PL/M-286 and PL/M-386, the RAM setting places the CONSTANT section within the DATA segment in all segmentation models (sizes).

For all of the microprocessors, the ROM setting places constants in the CODE segment. Under this setting, the INITIAL attribute on a variable produces a warning message. The dot operator is not advised for variable references under the ROM option because constants and variables will be relative to different segment registers. If SMALL is specified with the ROM control, then PL/M-86 and PL/M-286 pointers will be four bytes instead of two and PL/M-386 pointers will be six bytes instead of four. (See also Appendix F.)

If the keyword DATA is used in a PUBLIC declaration when compiling with the ROM control, DATA must also be used in the EXTERNAL declaration of program modules that reference it. However, no value list is then permitted because the data is defined elsewhere.

11.2.20 SAVE/RESTORE

Form	SAVE RESTORE
Default	None
Type	General

Discussion

With these controls the settings of certain general controls can be saved on a stack and then restored. The main usage of these controls is saving the controls before an included file and restoring them after inclusion of that file is complete. The controls whose settings are saved and restored are:

CODE/NOCODE
COND/NOCOND
LEFTMARGIN
LIST/NOLIST
OVERFLOW/NOOVERFLOW

The SAVE control saves all of these settings on a stack. The maximum capacity of this stack corresponds to the maximum nesting depth for the INCLUDE control (the maximum nesting depth is given in Appendix B).

The RESTORE control restores the most recently saved set of control settings from the stack.

11.2.21 SET/RESET

These are general controls. The SET control has the following general form:

Form SET(*switch_assignment_list*)

Where:

switch_assignment_list consists of one or more switch assignments separated by commas.

A switch assignment has the form:

switch

or

switch = value

Where:

switch	is a name which is formed according to the PL/M rules for identifiers. Note that a switch name exists only at the compiler control level, and therefore a switch can have the same name as an identifier in the program; no conflict is possible. Note however that no PL/M reserved word other than a predefined switch can be used as a switch name.
value	is a whole-number constant in the range 0 to 255. This value is assigned to the switch. If the value and the equal sign (=) are omitted from the switch assignment, the default value 0FFH (true) is assigned to a switch.

The following is an example of a SET control line:

```
$SET(TEST,ITERATION = 3)
```

This example sets the switch TEST to true (0FFH) and the switch ITERATION to 3. Switches do not need to be declared.

Figure 11-1 and 11-2 (see Section 11.1.10) are examples of a program that was compiled using the SET control.

The RESET control has the form:

RESET (*switch_list*)

Where:

switch_list consists of one or more switch names that have already occurred in SET controls.

Each switch in the switch list is set to false (0).

Default None

Type General

11.2.22 SMALL/COMPACT/MEDIUM/LARGE

The following sections describe the SMALL, COMPACT, MEDIUM, and LARGE controls (also called the segmentation controls).

11.2.22.1 SMALL

Form SMALL

Default SMALL for PL/M-86 and PL/M-386
LARGE PL/M-286

Type Primary

Discussion

PL/M-86

When modules compiled with the SMALL control are linked, the code sections from all modules are combined and are allocated space within one segment. The segment address for this segment is kept in the CS register. The constant, data, stack, and memory sections from all modules are allocated space within a second segment. The segment address for this second segment is kept in the DS register, with an identical copy in the SS register.

Therefore, the SMALL control may be used only if the total size of all code sections does not exceed 64K bytes, and the total size of all constant, data, stack, and memory sections does not exceed 64K bytes.

Because there is only one segment for code, the segment address for this segment (CS register) is never updated during program execution. Likewise, since there is only one segment for constants, data, stack, and memory sections, the segment address for this segment (DS and SS registers) is never updated (except when an interrupt occurs, as explained in Appendix G). Therefore, when any location is referenced, only a 16-bit offset is calculated and then used in conjunction with the appropriate segment address. POINTER values are therefore 16-bit values in the SMALL case, and this leads to the following restrictions.

1. POINTER variables cannot be initialized with, or assigned, whole-number constants. For example:

```
DECLARE RR POINTER INITIAL (2277);
                                        /* invalid under SMALL */
DECLARE SS POINTER;
SS = 100;                               /* invalid under SMALL */
```

2. The @ operator must not be used with a variable that was located at an absolute address specified by a whole-number constant. For example:

```
DECLARE JO BYTE AT (100), PO POINTER;
PO = @JO;                               /* invalid under SMALL */
```

   This restriction does not apply if the absolute address in the declaration is created by the @ operator with a variable, as in:

```
DECLARE UKE BYTE, NAGE POINTER;
DECLARE SKI BYTE AT (@UKE);
NAGE = @SKI;                                       /* valid */
```

3. The PUBLIC attribute must not be used with a variable located at an absolute address specified by a whole-number constant. As in restriction 2, this does not apply when @ is used:

```
DECLARE SHOMEN BYTE PUBLIC AT (100);         /* invalid */
DECLARE IKYO BYTE;
DECLARE SANKYO BYTE PUBLIC AT (@IKYO);         /* valid */
```

   This restriction arises because external variables are assumed to be in the DATA segment.

4. Restrictions 1 and 2 apply also to WORD variables when used as offset pointers and to the use of the dot operator.

5. The @ and dot operators cannot be used with variables based on SELECTOR. For example:

```
DECLARE SEL SELECTOR;
DECLARE R BASED SEL BYTE;
DECLARE PO POINTER;
PO = @R                                 /* invalid under SMALL */
```

6. If the built-in function SELECTOR$OF is used, it will always return the value of the DS register. If BUILD$PTR is used, it will ignore the SELECTOR expression (see Section 9.8).

Modules compiled with the SMALL control have three sections: code, data, and stack (see the OBJECT control). When these modules are linked, similar sections from each module are combined to form two segments: code and data. The maximum size of each segment for the 80286 microprocessor is 64K bytes. For the 80386 microprocessor, the maximum size of each segment is 4G bytes.

In the default SMALL case (RAM), the code sections from all modules are allocated space within the code segment, which is addressed relative to the CS register. Constants are combined with all the data and stack sections in the data segment. For the 80286 microprocessor, this segment is addressed relative to the DS register, with an identical copy in the SS register. For the 80386 microprocessor, this segment is addressed relative to the DS register, with an identical copy in the SS and ES registers. Specific to the 80386 microprocessor, none of the segment registers are changed during the course of program execution except ES, which is used to perform string operations, and FS and GS, which are used to address data exported by another subsystem. (Subsystems are described in Chapter 13.)

Therefore, the SMALL control can be used only if the total size of all code sections does not exceed 64K bytes for the 80286 microprocessor and 4G bytes for the 80386 microprocessor. Additionally, the total size of the constants plus all data and stack sections does not exceed the previously stated limits.

If the ROM control is used, the constants from all the modules are placed with the code in the code segment. The data segment then contains only the data and stack sections from all the modules.

Because only one code segment exists, its segment selector (the CS register) is never updated during program execution. (However, an interrupt will update the CS register.) Likewise, when RAM is used, only one segment exists for all constant, data, and stack sections. The segments' selectors (the DS and SS registers) are never updated (except when an interrupt occurs, as explained in Appendix G). Therefore, when any location is referenced, only a 16-bit offset for the 80286 microprocessor and a 32-bit offset for the 80386 microprocessor is calculated and used in conjunction with the appropriate segment selector. POINTER values in the SMALL (RAM) case are 16-bit values for the 80286 microprocessor and 32-bit values for the 80386 microprocessor.

The following restrictions must be observed:

1. Do not use the @ and dot operations with variables based on SELECTOR. For example:

```
DECLARE SEL SELECTOR;
DECLARE R BASED SEL BYTE;
DECLARE PO POINTER;
PO = @R;                          /* invalid under SMALL RAM */
```

2. Do not use the built-in function BUILD$PTR (see Section 9.6).

end
PL/M-286/386

PL/M-86/286

PL/M-80 Compatibility

The SMALL control is the most likely choice when recompiling a program written in PL/M-80. The compiler produces error messages to flag violations of any of the restrictions or to flag the use of the new reserved keywords (DWORD, INTEGER, REAL, POINTER, SELECTOR, and CAUSE$INTERRUPT) as programmer-defined identifiers. Otherwise, complete upwards compatibility is provided by PL/M-86 and PL/M-286.

end
PL/M-86/286

NOTE

Care must be used with the dot operator under conditions other than SMALL (RAM).

11.2.22.2 COMPACT

Form	COMPACT
Default	SMALL for PL/M-86 and PL/M-386 LARGE for PL/M-286
Type	Primary

Discussion

PL/M-86

A program compiled with the COMPACT control has four sections: code, data, stack, and memory. Each of these is the result of combining the same-type sections from all modules, and each has a maximum size of 64K bytes. The constant sections from all modules are grouped with the data segment unless the ROM control is used, which causes all constant sections to be merged into the CODE segment instead. Since the code, data, and stack segments are fully defined by the time the program is loaded, the segment base addresses in the CS and SS registers are never changed. (The DS register may change when an interrupt occurs, as explained in Appendix F.)

All code and a few prologue constants are addressed relative to CS. All data except absolute data (declared with the AT constant attribute) are addressed relative to DS. The stack is addressed relative to SS. ES is not initialized and can change during execution. References to any location require only a 16-bit offset address against these segment base addresses.

Do not declare PUBLIC variables AT an absolute location, as in:

```
    DECLARE ANVIL BYTE PUBLIC AT (100);
```

This restriction does not apply when the location within the AT attribute is formed with the @ operator. Therefore, DECLARE ANVIL BYTE PUBLIC AT (@HAMR) is valid. However, do not use the phrases @ MEMORY and MEMORY to define a PUBLIC variable.

end PL/M-86

PL/M-286/386

Modules compiled with the COMPACT control have three sections: code, data, and stack (see the OBJECT control). When these modules are linked, similar sections from each module are combined to form three segments: code, data, and stack. The maximum size of each segment is 64K bytes for the 80286 microprocessor and 4G bytes for the 80386 microprocessor.

In the default COMPACT case (RAM), the code sections from all modules are allocated space within the code segment, which is addressed relative to the CS register. Constants and all data sections are combined in the data segment, which is addressed relative to the DS register and, for the 80386 microprocessor, an identical copy is stored in the ES register. The stack is addressed relative to SS. Specific to the 80386 microprocessor, none of the segment registers are changed, except ES, which is used to perform string operations, as well as FS and GS, which are used to address data exported by another subsystem.

If the ROM control is used, the constants from all the modules are placed with the code in the code segment. The data segment then contains only the data sections from all the modules.

Since the code, data, and stack segments are fully defined by the time the program is loaded, the segment selectors in the CS and SS registers are never changed.

For the 80286 microprocessor, ES is not initialized and can change during execution. References to any location require only a 16-bit offset against these segment selectors.

Specific to the 80386 microprocessor, all six segment registers are initialized by the loader, with ES, FS, and GS initialized to DS. The DS and ES registers are also saved and reinitialized in each interrupt procedure prologue and epilogue to enable distinct interrupt environments. The FS and GS registers are volatile after initialization. References to any location require only a 32-bit offset against these segment selectors.

Observe the following restrictions when using COMPACT.

1. When an exported procedure is indirectly activated, a POINTER variable must be used in the CALL statement. For example:

```
$COMPACT(SUBSYS HAS MOD1, MOD2, MOD3; EXPORTS PROC)
MOD: DO
     DECLARE P POINTER, W WORD;
     PROC: PROCEDURE PUBLIC;
     .
     .
     .
     END PROC;
     P = @PROC; CALL P;            /* POINTER must be used */
     W = .PROC; CALL W;                        /* not allowed */
END MOD1;
```

PL/M-86/286

2. For the 8086 and 80286 microprocessors, when a procedure that is not exported is indirectly activated, a WORD variable must be used. Note that WORD variables do not range over the entire microprocessor address space, but are restricted to offsets within the current code segment. For example:

```
DECLARE P POINTER, W WORD;
LPROC: PROCEDURE;                                /* local */
.
.
.
END LPROC;
P = @LPROC; CALL P;                        /* not allowed */
W = .LPROC; CALL W;                    /* WORD must be used */
```

end PL/M-86/286

PL/M-386

3. For the 80386 microprocessor, when a procedure that is not exported is indirectly activated, an OFFSET variable must be used. Note that OFFSET variables do not range over the entire microprocessor address space, but are restricted to offsets within the current code segment. For example:

```
DECLARE P POINTER, O OFFSET;
LPROC: PROCEDURE;                                /* local */
.
.
.
END LPROC;
P = @LPROC; CALL P;                        /* not allowed */
O = .LPROC; CALL O;                  /* OFFSET must be used */
```

end PL/M-386

11.2.22.3 MEDIUM

Form	MEDIUM
Default	SMALL for PL/M-86 and PL/M-386 LARGE for PL/M-286
Type	Primary

For PL/M-86, modules compiled with the MEDIUM control have four sections: constant, data, stack, and memory. For PL/M-286, modules compiled with the MEDIUM control have three sections: code, data, and stack (see the OBJECT control). In the default MEDIUM case (RAM), the code section from each module makes up its own code segment. Thus, the total code space for each program may exceed 64K bytes, though the maximum size of any one code segment is limited to 64K bytes. At any moment during program execution, one code segment is the current segment, and its segment selector is kept in the CS register. This selector is updated whenever a PUBLIC or EXTERNAL procedure is activated, because a new code segment may be created as the result of a procedure activation.

All sections for all of the modules are combined in the data segment, which is addressed relative to the DS register (with an identical copy in the SS register) and is never changed (except when an interrupt occurs, as explained in Appendix G).

Specific to PL/M-286, if the ROM control is used, the constant section from each module is placed with the code in its code segment. The data segment then contains only the data and stack sections from all the modules.

Specific to PL/M-86, with the MEDIUM control, a POINTER value is a four-byte quantity containing a segment address and an offset.

Because the code segment selector may change during program execution, the following restrictions must be observed.

1. When a PUBLIC (or EXTERNAL, for PL/M-86) procedure is indirectly activated, a POINTER variable must be used in the CALL statement. For example:

```
DECLARE P POINTER, W WORD;
PROC: PROCEDURE PUBLIC;
.
.
.
END PROC;
P = @PROC; CALL P;     /* POINTER must be used for an indirect call */
W = .PROC; CALL W;                                      /* not allowed */
```

PL/M-86/286
(continued)

2. When a procedure that is not PUBLIC or EXTERNAL is indirectly activated, a WORD variable must be used. This is consistent with PL/M-80, but is not recommended for PL/M-86 and PL/M-286 programs because WORD variables do not range over the entire microprocessor address space. The WORD variables are restricted to offsets within the current code segment. For example:

```
DECLARE P POINTER, W WORD;
LPROC: PROCEDURE;                                   /* local */
   .
   .
END LPROC;
P = @LPROC; CALL P;                               /* invalid */
W = .LPROC; CALL W;                       /* valid, however */
                                          /* not recommended */
```

3. Specific to the 8086 microprocessor, a variable that is absolutely located (by using the AT attribute with a numeric constant) cannot have the PUBLIC attribute. The following example is invalid in PL/M-86:

```
DECLARE B BYTE PUBLIC AT(100);
```

end
PL/M-86/286

For PL/M-386, the MEDIUM control is provided for PL/M-86 and PL/M-286 compatibility. The MEDIUM control is interpreted exactly like the SMALL control. For more information, refer to the SMALL control entry in this chapter (Section 11.2.22.1).

11.2.22.4 LARGE

Form	LARGE
Default	SMALL for PL/M-86 and PL/M-386 LARGE for PL/M-286
Type	Primary

Discussion

For PL/M-86, a module compiled with the LARGE control has four sections: code (with constants unless RAM is specified; see Section 11.2.19), data, stack, and memory. For PL/M-286, a module compiled with the LARGE control has three sections: code (with constants if the ROM control is used), data (with constants if the RAM control is used), and stack.

For both the 8086 and the 80286 microprocessors, each section has a separate segment. Thus, the total space required for each section may exceed 64K, but the total size for each section from any one module is limited to 64K.

At any moment during program execution, one code segment and one data segment are current. Code and data segments are paired, so that the current code and data segments are always from the same module. The compiler implements this pairing by placing the segment selector from the data segment in a reserved location in the code section. During program execution, the segment selectors for the current code and data segments are kept in the CS and DS registers, respectively, and are updated whenever a PUBLIC or EXTERNAL procedure is activated, because this may involve new code and data segments.

The stack segment selector is kept in the SS register. Since the code segment selector may change during program execution, the following restrictions must be observed.

1. When a PUBLIC or EXTERNAL procedure is indirectly activated, a POINTER variable must be used in the CALL statement. For example:

```
DECLARE P POINTER, W WORD;
PROC: PROCEDURE PUBLIC;
.
.
.
END PROC;
P = @PROC; CALL P;                   /* POINTER must be used */
W = .PROC; CALL W;                           /* not allowed */
```

PL/M-86/286
(continued)

2. When a procedure that is not PUBLIC or EXTERNAL is indirectly activated, a WORD variable must be used. This is consistent with PL/M-80, but is not recommended for PL/M-86 and PL/M-286 because WORD variables do not range over the entire microprocessor address space. WORD variables are restricted to offsets within the current code segment. For example:

```
DECLARE P POINTER, W WORD;
LPROC: PROCEDURE;                                    /* local */
.
.
.
END LPROC;
P = @LPROC; CALL P;                            /* not allowed */
W = .LPROC; CALL W;                        /* WORD must be used */
```

end
PL/M-86/286

For PL/M-386, the LARGE control is provided for PL/M-86 and PL/M-286 compatibility. The LARGE control is interpreted exactly like the COMPACT control in most cases. For more information, refer to the COMPACT control entry in this chapter (Section 11.2.22.2). When the LARGE control is used in a PL/M-386 subsystem definition, it behaves differently from the COMPACT control. For more information about subsystems, see Chapter 13.

11.2.23 SUBTITLE

Form	SUBTITLE("*subtitle*")
Default	No subtitle
Type	General

Discussion

The subtitle character sequence (truncated on the right to fit, if necessary) is printed on the subtitle line of each page of listed output. Note that a subtitle specified on the invocation line must be enclosed in quotation marks.

The maximum length for subtitle is 60 characters, but a narrow pagewidth may restrict this number.

When a SUBTITLE control appears before the first noncontrol line in the source file, it causes the specified subtitle to appear on the first page and all subsequent pages until another SUBTITLE control appears.

A subsequent SUBTITLE control causes a page eject, and the new subtitle appears on the next page and all subsequent pages until the next SUBTITLE control.

11.2.24 SYMBOLS/NOSYMBOLS

Form	SYMBOLS NOSYMBOLS
Default	NOSYMBOLS
Type	Primary

Discussion

The SYMBOLS control specifies that a listing of all identifiers in the PL/M source program and their attributes is to be produced in the listing file.

The NOSYMBOLS control suppresses such a listing.

Note that the SYMBOLS control cannot override a NOPRINT control.

11.2.25 TITLE

Form	TITLE(*"title"*)
Default	TITLE (*"modulename"*)
Type	Primary

Discussion

The title character sequence, truncated on the right to fit, if necessary, is placed on the title line of each page of listing output. Note that the character sequence for a title must be enclosed in quotation marks when entered on the invocation line.

The maximum length for title is 60 characters, but a narrow pagewidth may restrict this number.

11.2.26 TYPE/NOTYPE

Form	TYPE NOTYPE
Default	TYPE
Type	Primary

Discussion

The TYPE control specifies that the object module is to contain information on the variable types output in symbol records. TYPE records provide a mechanism for promoting type compatibility between subprograms. This information may be used later for type checking when the program modules are combined, or by a debugger.

The NOTYPE control specifies that such type definitions are not to be placed in the object module.

PL/M-386

11.2.27 WORD32/WORD16

Form	WORD32 WORD16
Default	WORD32
Type	Primary

Discussion

The WORD32/WORD16 control determines how the compiler interprets the unsigned binary number and signed integer scalar types (as well as the built-ins that specify these data types) in the code being compiled.

When compiling PL/M-286, PL/M-86, or PL/M-80 source code with the PL/M-386 compiler, there are several points to consider before choosing WORD32 or WORD16. See Section 3.7 for a discussion of these points.

Table 11-2 lists the data types as interpreted by the compiler under WORD32 and WORD16. The WORD16 control does not mean creating PL/M-286 code, but rather that PL/M-386 data types are mapped to the equivalent PL/M-286 data type.

Table 11-2 WORD32/WORD16 Data Type Mapping

WORD32 (native)	WORD16 (mapping)	Number of Bits
Unsigned Binary Number Data Types		
BYTE	BYTE, HWORD	8
HWORD	WORD	16
WORD	DWORD	32
DWORD, QWORD	QWORD	64
Signed Integer Data Types		
CHARINT	SHORTINT, CHARINT	8
SHORTINT	INTEGER	16
INTEGER, LONGINT	LONGINT	32

Note that all built-ins that specify data types are different for WORD16. Table 11-3 lists the WORD32/WORD16 mapping for these built-ins. In PL/M-386, WORD16 keywords are mapped to the equivalent WORD32 keyword.

Table 11-3 WORD32/WORD16 Built-in Mapping

WORD32	WORD16
(type conversions)	(type conversions)
BYTE	BYTE, HWORD
HWORD	WORD
WORD	DWORD
DWORD, QWORD	QWORD
CHARINT	SHORTINT, CHARINT
SHORTINT	INTEGER
INTEGER	LONGINT
BLOCKINPUT	BLOCKINPUT
BLOCKOUTPUT	BLOCKOUTPUT
MOVB	MOVB, MOVHW
MOVRB	MOVRB, MOVRHW
FINDB	FINDB, FINDHW
FINDRB	FINDRB, FINDRHW
INPUT	INPUT, INHWORD
OUTPUT	OUTPUT, OUTHWORD
SKIPB	SKIPB, SKIPHW
SKIPRB	SKIPRB, SKIPRWH
CMPB	CMPB, CMPHW
SETB	SETB, SETHW
BLOCKINHWORD	BLOCKINWORD
BLOCKOUTHWORD	BLOCKOUTWORD
MOVHW	MOVW
MOVRHW	MOVRW
FINDHW	FINDW
FINDRHW	FINDRW
INHWORD	INWORD
OUTHWORD	OUTWORD
SKIPHW	SKIPW
SKIPRHW	SKIPRW
CMPHW	CMPW
SETHW	SETW

Table 11-3 WORD32/WORD16 Built-in Mapping (continued)

WORD32	WORD16
(type conversions)	(type conversions)
BLOCKINWORD	BLOCKINDWORD
BLOCKOUTWORD	BLOCKOUTDWORD
MOVW	MOVD
MOVRW	MOVRD
FINDW	FINDD
FINDRW	FINDRD
INWORD	INDWORD
OUTWORD	OUTDWORD
SKIPW	SKIPD
SKIPRW	SKIPRD
CMPW	CMPD
SETW	SETD

11.2.28 XREF/NOXREF

Form	XREF NOXREF
Default	NOXREF
Type	Primary

Discussion

The XREF control specifies that a cross-reference listing of source program identifiers is to be produced in the listing file.

The NOXREF control suppresses the cross-reference listing.

Note that the XREF control cannot override a NOPRINT control.

11.3 Program Listing

11.3.1 Sample Program Listing

During the compilation process, a listing of the source input is produced. Each page of the listing carries a numbered page-header that identifies the compiler, prints a time and date as designated by the host operating system, and optionally gives a title and a subtitle, and/or a date (see Figure 11-7).

The first part of the listing contains a summary of the compilation, beginning with the compiler identification and the name of the source module being compiled. The next line names the file receiving the object code. The next line contains the command used to invoke the compiler. The listing of the program itself is shown in Figure 11-7.

The listing contains a copy of the source input plus additional information. Two columns of numbers appear to the left of the source image. The first column provides a sequential numbering of PL/M statements. Error messages, if any, refer to these statement numbers. The second column gives the block nesting depth of the corresponding statement.

Lines included with the INCLUDE control are marked with an equal sign (=) just to the left of the source image. If the included file contains another INCLUDE control, lines included by this nested INCLUDE are marked with an =1. For yet another

```
PL/M-86 COMPILER   Stack Module                               mm/dd/yy hh:mm:ss  PAGE 1

system-id PL/M-86 Vx.y COMPILATION OF MODULE STACK
OBJECT MODULE PLACED IN stack.obj
COMPILER INVOKED BY: plm86 stack.src CODE XREF TITLE(Stack Module)

   1          STACK: DO;
                       /* This module implements a BYTE stack with push and pop */
   2   1        DECLARE S(100) BYTE,                                 /* Stack Storage */
                        T BYTE PUBLIC INITIAL(-1);                     /* Stack Index */
   3   1        PPUSH: PROCEDURE (B) PUBLIC;             /* Pushes B onto the stack */
   4   2          DECLARE B BYTE;
   5   2          S(T:=T+1) = B;                          /* Increment T and store B */
   6   2        END PPUSH;
   7   1        PPOP: PROCEDURE BYTE PUBLIC;/* Returns value popped from stack */
   8   2          RETURN S((T:=T-1)+1);             /* Decrement T, return S(T+ 1) */
   9   2        END PPOP;
  10   1      END STACK;                                          /* Module ends here */
```

Figure 11-7 Program Listing

level of nesting, =2 is used to mark each line, and so forth up to the compiler's limit of nesting levels (see Appendix B). These markings make it easy to see where included text begins and ends.

Should a source line be too long to fit on the page in one line, it is continued on the following line. Such continuation lines are marked with a hyphen (-) just to the left of the source image.

The CODE control can be used to obtain the assembly code produced in the translation of each PL/M statement. Figure 11-8 shows the assembly code listing for the program given in Figure 11-7. This code listing appears in six columns of information in a pseudo-assembly language format:

1. Location counter (hexadecimal notation)
2. Resultant binary code (hexadecimal notation)
3. Label field
4. Opcode mnemonic
5. Symbolic arguments
6. Comment field

Not all six of the columns will appear on all lines of the code listing. Compiler generated labels (e.g., those that mark the beginning and ending of a DO WHILE loop) are preceded by an AT sign (@). The comments appearing on PUSH and POP instructions indicate the stack depth associated with the stack instruction.

11.3.2 Symbol and Cross-Reference Listing

If specified by the XREF or SYMBOLS control, a summary of all identifier usage appears following the program listing. Figure 11-9 shows the cross-reference listing of the program given in Figure 11-7.

Depending on whether the SYMBOLS or XREF control was used to request the identifier usage summary, five or seven types of information are provided in the symbol or cross-reference listing. They are as follows:

1. Statement number where the identifier was defined.
2. Relative address associated with the identifier.
3. Size of the object identified (in bytes).
4. The identifier.

```
PL/M-86 COMPILER   Stack Module                  mm/dd/yy hh:mm:ss   PAGE 2
                    ASSEMBLY LISTING OF OBJECT CODE
                                                ; STATEMENT # 3

                        PPUSH        PROC NEAR
          0000   55                PUSH     BP
          0001   8BEC              MOV      BP,SP
                                                ; STATEMENT # 5

          0003   8A1E6400          MOV      BL,T
          0007   FEC3              INC      BL
          0009   881E6400          MOV      T,BL
          000D   B700              MOV      BH,0H
          000F   8A4604            MOV      AL,[BP].B
          0012   88870000          MOV      S[BX],AL
                                                ; STATEMENT # 6

          0016   5D                POP      BP
          0017   CA0200            RET      2H
                        PPUSH        ENDP
                                                ; STATEMENT # 7

                        PPOP        PROC NEAR
          001A   55                PUSH     BP
          001B   8BEC              MOV      BP,SP
                                                ; STATEMENT # 8

          001D   8A1E6400          MOV      BL,T
          0021   FECB              DEC      BL
          0023   881E6400          MOV      T,BL
          0027   FEC3              INC      BL
          0029   B700              MOV      BH,0H
          002B   8A870000          MOV      AL,S[BX]
          002F   5D                POP      BP
          0030   C3                RET
                                                ; STATEMENT # 9

                        PPOP ENDP
```

Figure 11-8 Code Listing

```
PL/M-86 COMPILER    Stack Module                          mm/dd/yy hh:mm:ss   PAGE 3
                    CROSS-REFERENCE LISTING

DEFN   ADDR   SIZE  NAME, ATTRIBUTES, AND REFERENCES
----  ------  ----  --------------------------------

   3  0004H      1  B. . . . . . . . . . . . BYTE IN PROC(PPUSH) PARAMETER AUTOMATIC  4 5
   7  001AH     23  PPOP . . . . . . . . . . PROCEDURE BYTE PUBLIC STACK=0002H
   3  0000H     26  PPUSH. . . . . . . . . . PROCEDURE PUBLIC STACK=0004H
   2  0000H    100  S. . . . . . . . . . . . BYTE ARRAY(100)    5* 8
   1  0000H         STACK. . . . . . . . . . MODULE STACK=0000H
   2  0064H      1  T. . . . . . . . . . . . BYTE PUBLIC INITIAL     5 5* 8 8*
```

Figure 11-9 Cross-Reference Listing

5. Attributes of the identifier (including expansion for LITERALLYs and scoping information for local variables and parameters). For PL/M-386, these attributes reflect the WORD32/WORD16 terminology of the source file.

6. Statement numbers where the identifier was referenced (XREF control only).

7. Statement numbers where the identifier was assigned a value (XREF control only).

Notice that a single identifier can be declared more than once in a source module (i.e., an identifier defined twice in different blocks). Each such unique object, even though named by the same identifier, appears as a separate entry in the listing.

The address given for each object is the location of that object relative to the start of its associated section. The object's attributes determine which section is applicable.

Identifiers in the SYMBOLS or XREF listing are given in alphabetical order with the following exception: members of structures are listed, in order of declaration, immediately following the entry for the structure itself. Indentation is used to differentiate between these entries.

The XREF listing differentiates between items 6 and 7 by adding the asterisk (*) character to statement numbers where a value is assigned. For example, if statement 17 read:

```
I = I + 1;
```

the list of statement numbers for I would include 17 and 17*, indicating a reference and an assignment in statement 17.

The AUTOMATIC attribute indicates that the identifier was declared as a parameter or as a local variable in a REENTRANT procedure and therefore is allocated dynamically on the stack.

11.3.3 Compilation Summary

Following the listing (or appearing alone if NOLIST is in effect) is a compilation summary. Eight pieces of information are provided:

- Code area size gives the size in bytes of the code section of the output module (not including constants, if any).
- Constant area size gives the size in bytes of the constant area. The constant area will be included with either the code or data section in the output module, depending on the specified compiler controls.
- Variable area size gives the size in bytes of the data section of the output module (not including constants, if any).
- Maximum stack size gives the size, in bytes, of the stack section allocated for the output module.
- Lines read gives the number of source lines processed during compilation.
- Program warnings give the number of warning messages issued during compilation.
- Program errors give the number of error messages issued during compilation.
- Dictionary summary gives the actual memory and disk space used by the dictionary during compilation.

Figure 11-10 is an example of the compilation summary.

```
MODULE INFORMATION:

     CODE AREA SIZE     = 0031H     49D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0065H    101D
     MAXIMUM STACK SIZE = 0004H      4D
     14 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

DICTIONARY SUMMARY:

     4KB MEMORY USED
     0KB DISK SPACE USED

END OF PL/M-86 COMPILATION
```

Figure 11-10 Compilation Summary

Sample Program 12

Tabs for
452161-001

12 SAMPLE PROGRAM CONTENTS

12 SAMPLE PROGRAM

12.1 Introduction

This chapter discusses a sample program consisting of three modules named FREQ, OPEN, and PRINT. The purpose of this program is to illustrate the use of the PL/M language. The program is written in PL/M-86, compiled with the PL/M-86 compiler, and combined with the LINK86 utility.

The program takes an input file, counts the uppercase and lowercase alphabetic characters, and determines the percentage of use for each character. This is printed either to the screen or, if one is specified, to an output file. The program's output lists the number of times each character is used (for uppercase, for lowercase, and for both uppercase and lowercase), and the percentage of use for each character. The source program listings are shown in Figures 12-1 through 12-3.

In addition to the main program modules (FREQ, OPEN, and PRINT), this program also has two include files. The include files, defns.inc and udi.inc (see Figures 12-4 and 12-5), contain definitions that are used in the program modules. The defns.inc include file consists of global variable definitions. The udi.inc include file consists of the universal development system interface (UDI) definitions. The UDI definitions are used for operating system interface (e.g., file manipulation). Figure 12-6 is an example of the program output.

The following sections describe the source code in each of the program modules. The line numbers in the figures are not part of the source code. The line numbers have been added to simplify the discussion of the source code.

12.2 FREQ Module

FREQ is the main module. The source code is shown in Figure 12-1. Note the line numbers in the figure. These are not part of the source code, nor are they the line numbers that the compiler would assign. The line numbers have been added to simplify the discussion of the source code.

The program lines that begin with a dollar sign ($) are compiler control lines. Lines that begin with a dollar sign instruct the compiler and are not part of the source program. In any position other than the first character (or the position specified with the LEFTMARGIN control), the dollar sign is an insignificant character and can be used as a separator to simplify the reading of variable names.

```
$DEBUG PW(75)
freq:DO;

 $INCLUDE (defns.inc)

$NOLIST
/*** LIST of UDI procedures is in OPEN.PLM ***/
$INCLUDE (udi.inc)
$LIST

open$files:PROCEDURE EXTERNAL;
       END open$files;

print$stats:PROCEDURE(arr$ptr, arr$len) EXTERNAL;
       DECLARE arr$ptr POINTER;
       DECLARE arr$len WORD;
       END print$stats;

       DECLARE buf(80) BYTE;
       DECLARE console CONNECTION EXTERNAL;
DECLARE i BYTE;
DECLARE infile CONNECTION EXTERNAL;
DECLARE lfreq(26) Freq_Struc;
DECLARE num$read BYTE;
DECLARE outfile CONNECTION EXTERNAL;
DECLARE quit$time BYTE INITIAL(False);
DECLARE status WORD;
DECLARE total WORD PUBLIC INITIAL (0);
$EJECT

main:

CALL open$files;

CALL init$real$math$unit;

DO i = 0 to LENGTH(lfreq);
   lfreq(i).let.low = 0;
   lfreq(i).let.up = 0;
   lfreq(i).percent = 0.0;
   END;
```

Figure 12-1 Source Code for FREQ Module

```
33                                                  /*** Now, read the files ***/

34 read$file:DO WHILE (NOT quit$time);
35    num$read = dq$read(infile,@buf,LENGTH(buf),@status);
36    IF num$read <> LENGTH(buf) THEN quit$time = True;

37    DO i = 0 to num$read;
38       total = total + 1;    /*** Total keeps track of ALL characters ***/
39                                       /*** read, not just the letters. ***/
40       sh_which_letter:IF (buf(i) >= 'A' AND buf(i) <= 'Z') THEN
41          lfreq(buf(i)-'A').let.up = lfreq(buf(i)-'A').let.up + 1;
42       ELSE IF (buf(i) >= 'a' AND buf(i) <= 'z') THEN
43          lfreq(buf(i)-'a').let.low = lfreq(buf(i)-'a').let.low + 1;

44    END;                                  /*** Loop i = 0 to num$read ***/
45    read$file:END;

46 stats:
47    CALL print$stats(@lfreq,LENGTH(lfreq));
48    CALL dq$exit(0);

49 END freq;
```

Figure 12-1 Source Code for FREQ Module (continued)

Line 1 specifies the DEBUG control and the pagewidth. The DEBUG control instructs the compiler to collect debug information such as the statement number and relative address of each source program module (see the DEBUG/NODEBUG entry in Chapter 11). PW(75) specifies an output page 75 characters wide.

Line 2 names the module and establishes the beginning of the module's DO block. As stated in Chapter 1, a module must begin with a labeled DO statement and end with an END statement.

Lines 3 through 6 specify the include files to be used in the program module. Line 4 indicates to the compiler to not list anything until the LIST control is encountered, which happens at line 7.

Line 5 is a user comment and will not be interpreted by the compiler. User comment lines begin with a slash/asterisk (/*) combination and end with an asterisk/slash (*/) combination.

Lines 8 through 23 are the procedure and variable declarations used in the FREQ module. Note the EXTERNAL declarations in lines 8 through 13. These procedures are declared EXTERNAL, which means that the procedure is defined in another module. The calling module must declare the procedure as EXTERNAL. The module in which these procedures are defined must declare the procedures as PUBLIC.

The variable declarations (see lines 15, 17, and 20) are also EXTERNAL. The same rules apply for variables as for procedures. The calling module must declare the variable as EXTERNAL and the defining module must declare the variable as PUBLIC. If the variable definition is included in the calling module, the definition must be identical to the definition in the declaring module.

Line 18 declares the lfreq structure, which is declared in the defns.inc file (see Figure 12-4). Line 21 declares quit$time as a variable (with the INITIAL attribute) of type BYTE. In an initialization, the initialization attribute must be placed after the variable attributes. In line 23, total is declared as a variable of type BYTE. Note also the PUBLIC declaration. This indicates that this variable can be used by other modules within the program (if it is declared EXTERNAL within the module which uses it).

Line 24 specifies the beginning of a new page (used when the program listing is printed).

The program begins at line 25. Line 26 calls the open$files procedure (declared as EXTERNAL in line 8). This procedure opens the input file, and if one is specified, the output file. Line 27 calls the compiler built-in procedure, init$real$math$unit. This call is required to initialize the REAL math facility (the numeric data extension) for subsequent operations.

Lines 28 through 32 consist of more initializations. These lines set (or reset) the values of the structure variable used in the module. Freq_struc is an array of nested structures (see Chapter 4). Freq_struc is a 26 element array (one element for each letter in the alphabet). Each element of the freq_struc array contains the let structure, which consists of a letter and a percent. Nested within the let structure is another structure (low and up). This structure holds the count of uppercase and lowercase characters. To see how freq_struc is declared, refer to Figure 12-4.

Lines 34 through 45 show an example of a nested DO block. With PL/M, DO blocks can be nested up to 18 levels. Line 37 begins a second DO block within the DO block that begins at line 34. The DO block nested within the first DO block ends at line 44. The first DO block ends at line 45.

Lines 34 through 36 use the UDI function, dq$read, to read from a file (infile). A specified number of characters are read from the file into an array. The array is buf and the number of characters read is LENGTH(buf). The value of buf was set in line 14. LENGTH is a built-in function (see Chapter 11) that returns the number of elements in an array. The UDI function, dq$read, returns the number of characters read (num$read) and an error code (status). For more information about UDI functions, refer to the run-time support manual.

The nested loop (lines 37 through 44) keeps totals for all the characters read, the uppercase letters read, and the lowercase characters read. This entire loop repeats until the number of characters read in from the input file is less than 80 (this indicates that the input file is empty).

Line 47 calls the external procedure print$stats. This procedure is defined in the PRINT module. Line 48 calls a UDI procedure, dq$exit. Finally, line 49 ends the FREQ module.

12.3 OPEN Module

The OPEN module takes care of the majority of the file-handling procedures for the program. This module makes extensive use of the UDI procedures provided by the run-time support library. The source code is shown in Figure 12-2. Note that the line numbers in the figure are not part of the source code, nor are they the line numbers that the compiler would assign. The line numbers have been added to simplify the discussion of the source code.

```
 1 $DEBUG PW(75)

 2 open:DO;
 3 $NOLIST
 4 $INCLUDE (defns.inc)
 5 $LIST

 6 $EJECT
 7 $INCLUDE(udi.inc)
 8 $EJECT
 9 DECLARE console CONNECTION PUBLIC;
10 DECLARE infile CONNECTION PUBLIC;
11 DECLARE outfile CONNECTION PUBLIC;

12 DECLARE NeedFile(*) BYTE INITIAL('Enter input file name: ');
13 DECLARE OpenError(*) BYTE INITIAL ('Error opening input file',CR,LF);
14 open$files:PROCEDURE PUBLIC;
```

Figure 12-2 Source Code for OPEN Module

```
    DECLARE delim BYTE;
    DECLARE console$in CONNECTION;
    DECLARE buffer(80) BYTE;
    DECLARE status WORD;
    DECLARE in$buf(81) BYTE;
    DECLARE i BYTE;
    DECLARE num$read BYTE;
    console = dq$create(@(4,':CO:'),@status);
    CALL dq$open(console,WriteOnly,0,@status);
    /*** Process the command line, it consists of three parts,
         1) the program name (lf.exe)
         2) the input file name, if this is not present then
            ask for it
         3) the output file name, if this is not present then
            the output goes to the console                           ***/
                                           /*** Read past the program name ***/
    delim = dq$get$argument(@buffer,@status);
                                       /*** Find out name of the input file ***/
    IF delim = CR THEN
       DO;
                                  /*** No input file specified, ask for it ***/
       CALL dq$write(console,@NeedFile,LENGTH(NeedFile),@status);
       console$in = dq$attach(@(4,':CI:'),@status);
       CALL dq$open(console$in,ReadOnly,0,@status);
       sch001:num$read = dq$read(console$in,@in$buf,LENGTH(in$buf),@status);
       CALL dq$close(console$in,@status);
                      /*** Convert the read in buffer to the infile buffer ***/
       sh_infile:buffer(0) = num$read;
       DO i = 0 to num$read;
          IF (in$buf(i) <> CR) AND (in$buf(i) <> LF)
             THEN buffer(i+1) = in$buf(i);
          ELSE
             buffer(0) = buffer(0) - 1;  /*** adjust count for CR/LF ***/
          END;                   /*** end of DO loop to Convert buffer ***/
       END;
    ELSE
       delim = dq$get$argument(@buffer,@status);
                                  /*** END; get file name to process ***/
                                                  /*** Open input file ***/
```

Figure 12-2 Source Code for OPEN Module (continued)

```
54    infile = dq$attach(@buffer,@status);
55    CALL dq$open(infile,ReadOnly,2,@status);
56    IF status <> E$OK THEN DO;
57       CALL dq$write(console,@OpenError,LENGTH(OpenError), @status);
58       CALL dq$exit(1);
59       END; /** status is not ok **/

60                /*** Find out if an output file was specified, if so, ***/
61                      /*** open it, if not use the console output   ***/
62    IF delim = CR THEN
63          outfile = console;
64    ELSE DO;
65       delim = dq$get$argument(@buffer,@status);
66       outfile = dq$create(@buffer,@status);
67       CALL dq$open(outfile,WriteOnly,2,@status);
68       END;
69 END open$files;

70 END open;
```

Figure 12-2 Source Code for OPEN Module (continued)

Line 1 instructs the compiler to collect debug information and sets the page width for printed output (see Section 12.1). Line 2 names the module and establishes the beginning of the module's DO block. Lines 3 through 8 specify the inclusion of the program's include files, turn the listing function on and off, as well as specify a few new pages for printed output ($EJECT).

Lines 9 through 11 define and declare some PUBLIC variables. Because these variables are declared PUBLIC, they can be used in another module. The calling module must declare the variable as EXTERNAL. The variable definition is included in the calling module, and it is the same as the definition in the defining module.

Lines 12 and 13 are error messages to be used by the OPEN module if the necessary information is not included in the invocation line (which causes an error). Note the use of the asterisk in each of these lines. The asterisk is used as an implicit dimension specifier. The implicit dimension specifier can be used when either the size of the array is unknown or insignificant. In this instance, the size of the array is unknown. The use of the implicit dimension specifier in lines 12 and 13 specifies that the NeedFile array and the OpenError array will have the same number of elements as the value list (the number of characters in the message).

Line 14 begins the open$files procedure. This procedure is declared as public (it is called by the FREQ module) and continues until the end of the module (line 69).

Lines 22 and 23 get and open a connection with the console using predefined UDI procedures. Note the use of the @ operator in these two lines. The first time the @ operator is used in line 22, it is used to allocate storage for the constants 4 and :CO:. The remaining uses of the @ operator are for location references. This means that the value of the reference (e.g., the value of @status) is the actual run-time location of the variable.

Lines 24 through 52 use the UDI procedure, dq$get$argument, to parse the input line. Line 31 gets the first part of the command line, as well as the delimiter used to separate this part of the command line from the next part (if there is any). Line 33 tests the delimiter. If the delimiter is a carriage return then lines 34 through 49 are processed. Lines 34 through 49 request a file name. If the delimiter is not a carriage return then dq$get$argument is called again. This routine is also called by lines 60 through 68 to determine whether the program output should go to a file or to the console.

Line 31 passes the invocation line to the following IF/THEN/ELSE construct (lines 33 through 51). The IF/THEN/ELSE construct checks for an input file name. If no input file is specified, line 36 uses the NeedFile string declared in line 12. This prompts the user to enter an input file name. If no input file name is specified in response to the prompt, the program aborts. Otherwise, the string is converted as discussed in the preceding paragraph.

Lines 43 through 48 convert the file name to a UDI call.

Lines 50 and 51 are the ELSE clause of delim = CR.

Lines 53 through 59 open the input file. Lines 60 through 68 open an output file, if one is specified. Otherwise, the program data is sent to the console.

Line 69 is the end statement for the open$files procedure and line 70 is the end statement for the OPEN module.

12.4 PRINT Module

The PRINT module performs the program calculations and prints the information (either to the console or to the specified output file). The source code is shown in Figure 12-3. Note the line numbers in the figure. These are not part of the source code, nor are they the line numbers that the compiler would assign. The line numbers have been added to simplify the discussion of the source code.

```
 1 $DEBUG PW(75)
 2 print:DO;

 3 $NOLIST
 4 $INCLUDE (defns.inc)
 5 $INCLUDE (udi.inc)
 6 $LIST

 7 DECLARE BLANK$OUT$LINE LITERALLY
 8     'DO j = 0 TO LENGTH(line);line(j) = SPACE;END';
 9 DECLARE LETTER LITERALLY '3';
10 DECLARE LOWER  LITERALLY '24';
11 DECLARE PCT LITERALLY '33';
12 DECLARE SUM    LITERALLY '8';
13 DECLARE UPPER  LITERALLY '16';

14 DECLARE outfile CONNECTION EXTERNAL;
15 DECLARE topline(*) BYTE INITIAL
16      ('LETTER TOTAL UPPER LOWER % ',CR,LF);
17              /** (   A      00000    00000    00000    000.0      ***/
18              /** ( 123456789 123456789 123456789 123456789 123456789 ***/
19 DECLARE total WORD EXTERNAL;
20 DECLARE total$str (5) BYTE INITIAL ('TOTAL');

21 int2asc:PROCEDURE(number,stg$ptr,count) BYTE;
22     DECLARE number WORD;
23     DECLARE stg$ptr POINTER;
24     DECLARE count BYTE;

25     DECLARE i BYTE, j BYTE;
26     DECLARE max DWORD;
27     DECLARE string BASED stg$ptr(1) BYTE;
28     DECLARE tmpstg(10) BYTE;

29     max = 1;
30     DO i = 1 TO count;
31        max = 10 * max;
32        END;
33     max = max - 1;
```

Figure 12-3 Source Code for PRINT Module

```
   DO i = 0 TO LAST(tmpstg);
      tmpstg(i) = SPACE;
      END;
   IF number <= max THEN DO;
      i = 0;
      loop:
         tmpstg(i) = (number MOD 10) + '0';
         i = i + 1;
         number = number/10;
      IF number 0 THEN GOTO loop;
      DO j = 0 TO count;
         string(count-j) = tmpstg(j);
         END;
      END;
   ELSE DO;
         DO i = 0 to count;
         string(i) = '*';
         END;
   END;
   RETURN(i);
   END int2asc;
real2asc:PROCEDURE(number,stg$ptr,count);
   DECLARE number REAL;
   DECLARE stg$ptr POINTER;
   DECLARE count WORD;
   DECLARE i BYTE, j BYTE;
   DECLARE int$len BYTE;
   DECLARE string BASED stg$ptr(1) BYTE;
   DECLARE tmpnum DWORD;
   DECLARE tmpstg(10) BYTE;
   /*** Convert the number to an INTEGER to convert it, assume one ***/
                                                  /*** decimal place ***/
   tmpnum = DWORD(number*10.0);
   int$len = int2asc(tmpnum,@tmpstg,LAST(tmpstg));
   IF int$len = 1 THEN DO;     /*** Handle the case where the number ***/
                                               /*** is less than 1.0 ***/
      int$len = 2;
      tmpstg(LAST(tmpstg)-1) = '0';
      END;
```

Figure 12-3 Source Code for PRINT Module (continued)

```
 73    DO i = 0 TO int$len-2;
 74       string(count-i) = tmpstg(LAST(tmpstg)-i);
 75       END;
 76    string(count-int$len) = '.';
 77    string(count-int$len-1) = tmpstg(LAST(tmpstg)-int$len+1);
 78    END real2asc;
 79      $EJECT
 80     print$stats:PROCEDURE (arr$ptr, arr$len) PUBLIC;
 81        DECLARE arr$ptr POINTER;
 82        DECLARE arr$len WORD;
 83        DECLARE array BASED arr$ptr(1) Freq_Struc;
 84        DECLARE i BYTE, j BYTE;
 85        DECLARE line(50) BYTE;
 86        DECLARE status WORD;
 87        DECLARE tmp BYTE;
 88        DECLARE ii BYTE;
 89        call dq$write(outfile,@topline,LENGTH(topline),@status);
 90        printlines:DO ii = 0 TO arr$len-1;
 91           BLANK$OUT$LINE;
 92           line(LETTER) = ii + 'A';
 93                     /*** Get the total and convert number to ascii ***/
 94     tmp = int2asc( (array(ii).let.low + array(ii).let.up), @line(SUM),5);
 95     tmp = int2asc (array(ii).let.low, @line(LOWER),5);
 96     tmp = int2asc (array(ii).let.up, @line(UPPER),5);
 97     array(ii).percent = REAL((array(ii).let.low) + (array(ii).let.up)) /
 98                              REAL(total) * 100.0;
 99      CALL real2asc (array(ii).percent, @line(PCT),5);
100      line(LAST(line)-1) = CR;
101      line(LAST(line)) = LF;
102      CALL dq$write(outfile,@line,LENGTH(line),@status);
103    END printlines;                                 /*** print loop ***/
104    BLANK$OUT$LINE;
105    DO i = 0 TO LAST(total$str);
106      line(LETTER-2"i)= total$str(i);
107    END;
108    tmp = int2asc( total, @line(SUM),5);
109    call dq$write(outfile,@line,LENGTH(line),@status);
110 END print$stats;
111 END print;
```

Figure 12-3 Source Code for PRINT Module (continued)

Line 1 instructs the compiler to collect debug information and sets the page width for printed output (see Section 12.1). Line 2 names the module and establishes the beginning of the module's DO block. Lines 3 through 6 specify the inclusion of the program's include files and turn the listing function on and off.

Lines 7 through 13 are a group of literally definitions. A literally definition is used to create an alternate name for a sequence of characters. Lines 7 and 8 declare BLANK$OUT$LINE as the alternate name for the DO loop used to blank out the output line buffer. Additionally, after line 13, the number 16 will reference upper (for uppercase character). This is a useful function to eliminate keystrokes, to make the program more readable, and to declare quantities that may be fixed in one module, but subject to change in another module.

Lines 14 through 20 contain more declarations, as well as the header string for the output (line 16).

Lines 21 through 54 perform an integer to ASCII translation. Lines 55 through 78 perform a real number to ASCII translation.

Line 80 is the beginning of the print$stats procedure. The print$stats procedure is called by the FREQ module, therefore it is declared PUBLIC in this module. Note the based variable in line 83. In this instance, the location of array is based on the address of arr$ptr, which is passed into the print$stats procedure. The size of the array is unknown (except through the parameter). The 1 enclosed in parentheses enables the use of arr$ptr as an array (any number can be used).

Line 89 calls a UDI procedure that writes to an external connection declared in the OPEN module. Note the use of BLANK$OUT$LINE in line 91.

Lines 90 through 103 are a DO loop that is repeated for each letter in the alphabet. For each character, the line(LETTER) array is filled with the letter, the total, the total uppercase, the total lowercase, and the percent. This information is then sent to the specified output device (the console or a file).

Lines 93 through 96 call the procedure to convert the total into ASCII characters. Lines 97 and 98 figure the percentage of use for each character. Line 99 calls the procedure to convert the percentage to ASCII characters. Lines 100 and 101 insert a carriage return and a line feed in the console display or in the output file.

Line 110 ends the print$stats procedure and line 111 ends the PRINT module.

12.5 Include Files

As stated in Section 12.1, there are two include files with this program (see Figures 12-4 and 12-5).

```
DECLARE DCL LITERALLY 'DECLARE';
DCL LIT        LITERALLY 'LITERALLY';

DCL CR         LITERALLY 'ODH';
DCL LF         LITERALLY 'OAH';

DCL True       LITERALLY 'OFFH';
DCL False      LITERALLY 'OOOH';

DCL Freq_Struc LITERALLY 'STRUCTURE (let STRUCTURE
                                        (low WORD, up WORD),
                                    percent REAL)';

DCL SPACE       LITERALLY 'O2OH';
```

Figure 12-4 Include File — defns.inc

Figure 12-4 is the defns.inc file. It contains definitions for terms used in common by all of the modules in the program (excluding the UDI definitions). Note the declaration of a structure in this include file (freq_struc). This structure is used in the PRINT module and the FREQ module. This structure declaration illustrates several levels of nesting. Structures can be nested up to 32 levels.

Figure 12-5 is the udi.inc file. It contains UDI definitions that are used throughout the modules. The UDI is a predefined set of procedure calls that enables use of operating system functions. For more information on UDI definitions, see the run-time support manual.

```
DECLARE CONNECTION literally 'WORD';
DECLARE ReadOnly LITERALLY '1';
DECLARE WriteOnly LITERALLY '2';
DECLARE E$OK LITERALLY '0H';

dq$attach:procedure (path$p,except$p) CONNECTION external;
     declare path$p pointer; declare except$p pointer;
     end dq$attach;

dq$close:procedure (aftn,exception$ptr) external;
   declare aftn CONNECTION, exception$ptr pointer;
   end dq$close;

dq$create:procedure (path$p,exception$ptr) CONNECTION external;
   declare (path$p,exception$ptr) pointer;
   end dq$create;

dq$exit:procedure ( completion$code) external;
                declare completion$code word;
                end dq$exit;

dq$get$argument:PROCEDURE (arg$ptr, ex$ptr) BYTE EXTERNAL;
     declare arg$ptr POINTER, ex$ptr POINTER;
     END dq$get$argument;

dq$open:procedure (aftn,mode,num$buf,exception$ptr) external;
   declare aftn CONNECTION, exception$ptr pointer;
   declare (mode,num$buf ) byte;
end dq$open;
dq$read:PROCEDURE(aftn,buf$ptr,count,ex$ptr) WORD EXTERNAL;
   declare aftn CONNECTION;
   declare buf$ptr POINTER;
   declare count WORD;
   declare ex$ptr POINTER;
   END dq$read;

dq$write:procedure (aftn,buffer,count,exception$ptr) external;
                declare aftn CONNECTION;
                declare count word;
                declare (buffer,exception$ptr) pointer;
                end dq$write;
```

Figure 12-5 Include File — udi.inc

Extended Segmentation Models 13

Tabs for
452161-001

13 EXTENDED SEGMENTATION MODELS CONTENTS

13 EXTENDED SEGMENTATION MODELS

13.1 Overview

Program segmentation is the division of a program into memory segments. It is a technique used to optimize the code produced by the compiler. The segmentation controls (SMALL, MEDIUM, COMPACT, and LARGE) manage program segmentation by defining the physical relationship in memory of a program's code, data, constants, and stack. They determine which (if any) segments get combined. For example, specifying the SMALL segmentation control for a program module locates all of the module's code, data, constants, and stack in two segments, CODE and DATA. When the program's modules are combined, sections from the separately compiled modules are combined into segments according to the specified segmentation controls. This optimizes code because references to locations in the same memory segment are more efficient.

Extended segmentation models are a super-set of the segmentation controls. The extended segmentation models (which consist of the SMALL, COMPACT, and LARGE subsystems) provide enhanced program speed and aid in the construction of large programs. An extended segmentation model consists of a number of subsystems. A subsystem is a collection of program modules that use the same segmentation controls. A program is made up of one or more subsystems. With subsystems, program modules that are compiled with different segmentation controls can be combined.

This chapter defines the use of extended segmentation models, and contains the following sections:

- Introduction
- Segmentation controls architecture overview
- Using subsystems
- Syntax
- Exporting procedures
- Large matrix example (PL/M-386 specific)

13.2 Introduction

Extended segmentation models provide the following programming advantages:

- Efficient use of memory.
- Access to the microprocessor's segmented architecture.
- Storage reduction for external references to pointers and code.
- Increased program execution speed for intersegment calls and data access.

Additionally, to simplify the development of large programs, the segmentation controls can be used to partition the program into a collection of related subsystems.

Partitioning a large program into a series of subsystems isolates code references within the same segment. The compiler processes each program module individually, assigning code, data and stack segments for each module (according to the specified segmentation control). As a source file is translated, the compiler generates a STACK segment for the program stack, as well as a DATA segment for the program data and a CODE segment for the program's executable code. When the program modules are combined, the CODE, DATA and STACK segments from all of the individual program modules are combined. Use of the segmentation controls ensures that the segment names generated by the compiler are combined according to the overall structure of the program.

A subsystem is either open or closed. An extended segmentation model can have only one open subsystem, but any number of closed subsystems.

An open subsystem does not have a name and claims the program modules that are not claimed by another subsystem. Effectively, a program that uses only the segmentation controls is an open subsystem. Modules can be added to the open system without having to change the subsystem definition.

A closed subsystem has a name and, optionally, a list of program modules used in the subsystem. To add a module to a closed subsystem, the subsystem definition must be changed.

13.3 Segmentation Controls Architecture Overview

The segmentation controls described in Chapter 11 define the physical relationship in memory of program code, data, constants, and stack during program execution. When a PL/M source file is compiled, an object module conforms to a particular extended segmentation model.

There are three extended segmentation models: SMALL, COMPACT, and LARGE. For the 8086 and the 80286 microprocessors, each segment can be as large as 64K bytes. For the 80386 microprocessor, each segment can be as large as 4G bytes.

There are two submodels within each model: RAM and ROM. Specifying RAM places the program constants in the DATA segment. Specifying ROM places the program constants in the CODE segment.

Tables 13-1 and 13-2 define the memory partitions and the placement of pointers in the various architectural models available with the segmentation controls. Table 13-1 shows how memory is partitioned. Table 13-2 defines the register addresses and the pointer values. Table 13-3 defines the register addresses and the pointer values for the 80386 microprocessor-specific ES register. Note that the POINTER variable value for the 80386 microprocessor using the SMALL ROM extended segmentation controls is 6 bytes.

Table 13-1 Segmentation Controls and Memory Partitions

Control	Segment Name		
	CODE	DATA	STACK
SMALL RAM	code	data constants stack	
SMALL ROM	constants code	data stack	
COMPACT RAM	code	data constants	stack
COMPACT ROM	constants code	data	stack
MEDIUM RAM*	separate CODE segment for each module's code	data constants stack	
MEDIUM ROM*	separate CODE segment for each module's code and constants	data stack	
LARGE RAM*	separate CODE segment for each module's code	separate DATA segment for each module's data and constants	stack
LARGE ROM*	separate CODE segment for each module's code and constants	separate DATA segment for each module's data	stack

*The 80386 microprocessor uses only the SMALL and COMPACT segmentation controls. For the segmentation controls (NOT subsystems), MEDIUM is equivalent to SMALL and LARGE is equivalent to COMPACT.

Table 13-2 Segmentation Controls, Register Addresses and Pointer Values

Control	Register Address			POINTER Variable Value
	CS	DS	SS	
SMALL RAM	CODE seg. Offset-reference relative to DS	DATA seg. Offset-reference relative to DS	DATA seg. Has same value as DS Offset-reference	2-byte offset only
SMALL ROM	CODE seg. Constant reference requires selector-offset containing CS value and offset within CODE segment Code reference requires offset-reference relative to DS	DATA seg. Offset reference	DATA seg. Has same value as DS Offset-reference	4-byte selector-offset for the 8086 and 80286; 6-byte selector-offset for the 80386
COMPACT RAM	CODE seg. Selector-offset reference	DATA seg. Selector-offset reference	STACK seg. Selector-offset reference	4-byte selector-offset
COMPACT ROM	CODE seg. Selector-offset reference	CODE seg. Selector-offset reference	STACK seg. Selector-offset reference	4-byte selector-offset
MEDIUM RAM	Current CODE seg. Selector-offset reference Updated when PUBLIC or EXTERNAL procedure is activated	DATA seg. Selector-offset reference	DATA seg. Selector-offset reference	4-byte selector-offset

Table 13-2 Segmentation Controls, Register Addresses and Pointer Values (continued)

Control	Register Address			POINTER Variable Value
	CS	DS	SS	
MEDIUM ROM	Current CODE seg. Selector-offset reference Updated when PUBLIC or EXTERNAL procedure is activated	DATA seg. Selector-offset reference	DATA seg. Selector-offset reference	4-byte selector-offset
LARGE RAM	Current CODE seg. Selector-offset reference Updated when PUBLIC or EXTERNAL procedure is activated	Current DATA seg. Selector-offset reference Updated when PUBLIC or EXTERNAL procedure is activated	STACK seg. Selector-offset reference	4-byte selector-offset
LARGE ROM	Current CODE seg. Selector-offset reference Updated when PUBLIC or EXTERNAL procedure is activated	Current CODE seg. Selector-offset reference Updated when PUBLIC or EXTERNAL procedure is activated	STACK seg. Selector-offset reference	4-byte selector-offset

The values given in Tables 13-1 and 13-2 are identical for the 80386 microprocessor. Additionally, the 80386 microprocessor has the ES register address. Table 13-3 states the values for the 80386 microprocessor-specific ES register.

Table 13-3 80386 Microprocessor-specific ES Register Segmentation Controls, Register Addresses and Pointer Values

Control	ES Register Address	POINTER Variable Value
SMALL RAM	DATA seg. Offset reference	4-byte offset only
SMALL ROM	DATA seg. Offset reference	6-byte selector offset
COMPACT RAM	DATA seg. Selector-offset reference	6-byte selector-offset
COMPACT ROM	DATA seg. Selector-offset reference	6-byte selector-offset

end PL/M-386

The SMALL RAM segmentation control is the most efficient. Because all of the code resides in one segment, jumps and calls are always within the same segment (intra-segment). However, the SMALL RAM segmentation control provides less protection and cannot be used to pass pointers to library procedures unless the library procedure is also a SMALL RAM model.

Use the COMPACT segmentation controls (COMPACT RAM and COMPACT ROM) for separate management of the code, data, and stack, or to improve segment-limit protection. To reference stack-based variables, the COMPACT segmentation controls use selector-offset references. This is less efficient than using offset-only references. However, data and constant references within a COMPACT segmentation control module are within the same segment (intra-segment). Note that if a COMPACT program must pass a data address to a procedure in a different subsystem, it must use a selector-offset reference.

PL/M-86/286

For the MEDIUM segmentation control, selector-offset references are used because there is more than one CODE segment. This enables pointer reference to procedures that reside in different CODE segments. Since there is more than one CODE segment, far calls and jumps (4 byte, selector-offset) are necessary. However, offset-only references can be used for the data, constants, and stack.

PL/M-86/286
(continued)

For the LARGE segmentation control, selector-offset references are used to enable pointers to procedures, data, and constants that reside in different CODE or DATA segments. Since there is more than one CODE segment, far calls and jumps are necessary.

end
PL/M-86/286

13.4 Using Subsystems

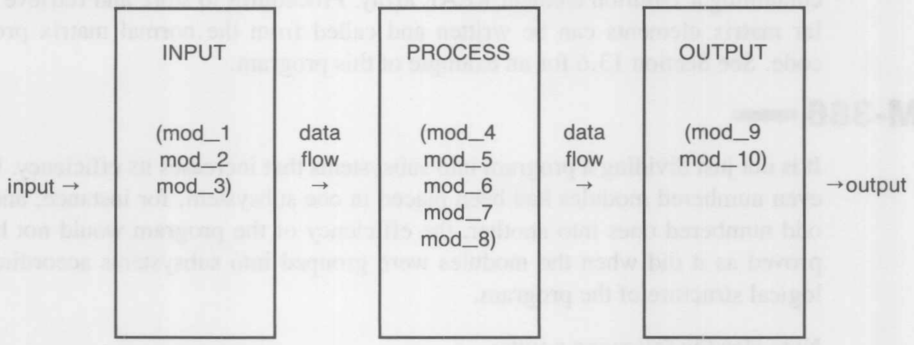
Subsystems offer an efficient way to manage programs with large amounts of data, to share data between program modules, and to communicate with other programs.

For example, subsystems are useful when several programmers are each writing a separate module for a highly structured program in which sharing data between modules is accomplished with parameter passing, by value only. To maintain the integrity of each section's data requires that each section have its own DATA segment. In this way, code in one module of the program cannot mistakenly destroy data belonging to another section of the program. In this instance, each module could be a COMPACT subsystem, with its own CODE and DATA segments.

As another example, a program performing I/O usually requires operating system support routines. In many cases, the operating system will operate at a higher protection level than the application program. Thus, operating system procedure calls are intersegment calls. The application program views the operating system as a separate subsystem. Usually, operating system interface libraries are supplied to application programmers; these libraries perform the inter-subsystem communication details. If a program needs to make a direct operating system call without using a presupplied library, the program itself must define the necessary subsystem environments at compile time.

It is usually more efficient to structure a large program with subsystems. With subsystems the code and data can be partitioned into manageable pieces bigger than one module. Within each subsystem, calls and jumps are near (2 byte offset), references can be offset only, and the data of each subsystem is protected from being overwritten by other subsystems. Calls and jumps between subsystems are still far, and references between subsystems need to be selector-offset. In general, a program's structure is such that it is possible to break the program into pieces with a minimum number of intersegment calls, jumps, and references.

For example, consider a program consisting of 10 modules, mod_1 through mod_10. Modules 1 through 3 deal with input and initial processing. Modules 4 through 8 do the main data processing. Modules 9 and 10 output the data. The following figure illustrates the structure of the program:



The total code space required by this program exceeds 64K bytes, and the total data space also exceeds 64K bytes. Thus, for the 8086 and the 80286 microprocessors, LARGE is the only segmentation control that could be used. The LARGE segmentation control provides each module with its own CODE and DATA segment. For this example, this results in a total of 21 segments (10 CODE, 10 DATA, and 1 STACK). For the LARGE segmentation control, all calls and jumps are far, and all intermodule references must be through selector-offset POINTERS.

If, for example, COMPACT subsystems are used instead of the LARGE segmentation control, modules 1 through 3 can form one subsystem, call it SUB_INPUT. Modules 4 through 8 can form subsystem SUB_PROCESS. Finally, modules 9 and 10 can form subsystem SUB_OUTPUT. The number of segments has been reduced to seven 3 CODE, 3 DATA, and 1 STACK). Since most of the calls, jumps, and references now take place within only one of the subsystems, the program is much more efficient. The only far calls and jumps, and the only selector-offset references needed are those in the interfaces between the subsystems.

PL/M-386

For the 80386 microprocessor, a typical program does not require subsystems. The code space of 4 gigabytes and the data space of 4 gigabytes is quite sufficient for most programs. However, consider a program that processes a large amount of data such as a 10x1,000,000,000 REAL matrix. A REAL scalar consists of 4 bytes, so the total memory needed is 40 billion bytes. Rows could be used to partition the matrix. Each row would be 4 billion bytes, which would fit into a single DATA segment.

PL/M-386
(continued)

Ten COMPACT subsystems (named ROW1, ROW2, etc.) could be created, each containing a 1-billion element REAL array. Procedures to store and retrieve particular matrix elements can be written and called from the normal matrix processing code. See Section 13.6 for an example of this program.

end
PL/M-386

It is not just dividing a program into subsystems that increases its efficiency. If all the even numbered modules had been placed in one subsystem, for instance, and all the odd numbered ones into another, the efficiency of the program would not have improved as it did when the modules were grouped into subsystems according to the logical structure of the program.

Note also the following points:

1. Not all subsystems must use the same segmentation control. For instance, if SUB_PROCESS in the preceding example is small enough, it could be a SMALL subsystem.
2. If a SMALL subsystem is mixed with subsystems using other segmentation controls, the main program must be in SMALL. This is because anything compiled in SMALL assumes that DS and SS are identical. This will be so only if the main program is SMALL. Notice that in this case, the STACK segment resulting from the COMPACT and LARGE subsystems will not be used, since the stack of the main program is in the combined DATA-STACK segment of the SMALL model.
3. SMALL RAM subsystems have the limitation that the SMALL segmentation control uses short (offset only) pointers. A SMALL RAM subsystem cannot receive a pointer from another subsystem, because it cannot save the selector portion. A SMALL RAM subsystem can, however, pass a pointer to a subsystem that is not SMALL RAM, because its own DS is known to it. (However, a SMALL RAM subsystem cannot pass a pointer (which points to a procedure), since DS is assumed as the selector to all pointers.)
4. For the 8086 and the 80286 microprocessors, using all LARGE subsystems has the same effect as using the LARGE segmentation control without subsystems. However, using a mixture of LARGE and other subsystems may be useful.
5. MEDIUM is a segmentation control only, not an extended segmentation model.

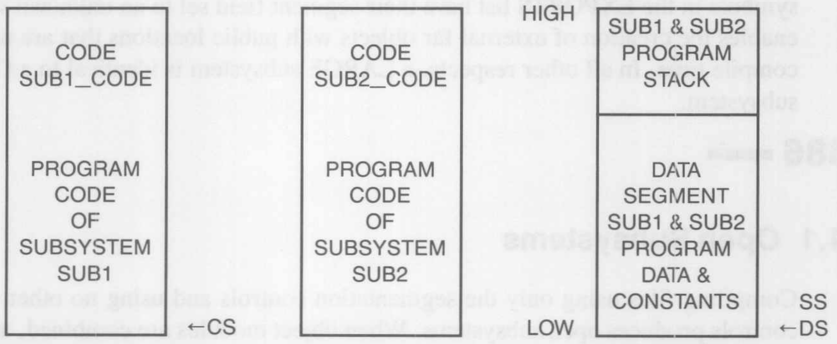
Section 13.3 describes the memory layouts of programs using the standard segmentation controls: SMALL/COMPACT/MEDIUM/LARGE. To understand the memory layouts of programs structured with subsystems, it is necessary to make the distinction between compiling modules and combining modules into a program.

The compiler compiles only one module at a time. When modules are combined into a program, many CODE, DATA and STACK segments, which were generated during separate compilations, are combined. When combining program modules, all segments with the same name are combined. The segmentation controls work by controlling the names of the segments generated by the compiler. This ensures that the segment names will be combined as desired when the modules are combined into a program.

The standard SMALL segmentation control causes the compiler to name the CODE segment CODE, and the DATA-STACK segment DATA. Since under the standard SMALL model all CODE segments have the same name, and all DATA-STACK segments have the same name, they are combined when the modules are combined.

A module belonging to a SMALL subsystem, on the other hand, takes the name of its CODE segment from the name of the subsystem. The name of its DATA-STACK segment is still DATA. Thus, a SMALL subsystem named SUB1 contains one CODE segment named SUB1_CODE, and one DATA-STACK segment named DATA. A SMALL subsystem named SUB2 contains one CODE segment named SUB2_CODE, and one DATA-STACK segment named DATA. When the program modules are combined, all segments with the same name are combined.

The memory layout of the loaded program containing the two subsystems SUB1 and SUB2 is as follows (it is assumed that both subsystems are SMALL RAM):


CODE
SUB1_CODE
PROGRAM CODE OF SUBSYSTEM SUB1
←CS
CODE
SUB2_CODE
PROGRAM CODE OF SUBSYSTEM SUB2
HIGH
SUB1 & SUB2 PROGRAM STACK
DATA SEGMENT SUB1 & SUB2 PROGRAM DATA & CONSTANTS
LOW
SS
←DS

Note that a program using the MEDIUM segmentation control is equivalent to a program in which each module is declared to be in a unique SMALL subsystem.

A module belonging to a COMPACT subsystem takes the name of its CODE segment and the name of its DATA segment from the subsystem name. So a COMPACT subsystem named SUB1 contains one CODE segment named SUB1_CODE, one

DATA segment named SUB1_DATA, and one STACK segment named STACK. A COMPACT subsystem named SUB2 contains one CODE segment named SUB2_CODE, one DATA segment named SUB2_DATA, and one STACK segment named STACK. The loaded program will contain five segments, two CODE segments, two DATA segments, and one STACK segment. Note that a program using the LARGE segmentation control is equivalent to a program in which each module is declared to be in a unique COMPACT subsystem.

A LARGE subsystem can be simulated by a COMPACT subsystem containing only one module. However, LARGE subsystems are useful for the following reason. A LARGE subsystem named SUB1, which contains the modules MOD1, MOD2, and MOD3, has three CODE segments named MOD1_CODE, MOD2_CODE, and MOD3_CODE, and three DATA segments named MOD1_DATA, MOD2_DATA, and MOD3_DATA. As usual, it contains one STACK segment named STACK. It is possible to use a LARGE subsystem instead of inventing names for many COMPACT subsystems, each containing only one module. Note that the segment name in the LARGE subsystem is derived from the module names and not from the subsystem name.

PL/M-386

For the 80386 microprocessor, the LARGE segmentation control is identical to the COMPACT segmentation control. However, there is a difference between LARGE and COMPACT subsystems. In a LARGE subsystem, the external definition of all symbols in the EXPORTS list have their segment field set to an unknown value. This enables the creation of external far objects with public locations that are unknown at compile time. In all other respects, a LARGE subsystem is identical to a COMPACT subsystem.

end PL/M-386

13.4.1 Open Subsystems

Compiling files using only the segmentation controls and using no other subsystem controls produces open subsystems. When object modules are combined, all modules created from compilations specifying a particular segmentation control are automatically combined. Segments are created according to the rules for the segmentation control. A list of modules belonging to an open subsystem is therefore not needed at compile time. Modules can be freely added to or deleted from an open subsystem at any time during program development.

Note that both RAM and ROM modules are combined into the single open subsystem. For a SMALL subsystem, be careful when combining RAM and ROM modules, particularly concerning the passing of pointer parameters and the accessing of constants not in the current module.

It is not possible to pass pointer parameters between SMALL RAM and SMALL ROM modules, because pointers are defined differently in each submodel. Also, it is not possible to directly reference constants defined in a ROM module from a RAM module, and vice versa, because RAM modules define constants to be in the data segment, and ROM modules define constants to be in the code segment.

In the COMPACT model, passing pointer parameters between RAM and ROM modules is not a problem, because pointers are always long. As in SMALL, the restriction on direct reference to constants applies.

PL/M-386

The names of the segments in both SMALL and COMPACT models are identical: CODE32 for the code segment, DATA for the data segment. This means that if SMALL and COMPACT modules are combined, they will also be combined to form a single open subsystem consisting of the CODE32, DATA, and STACK segments. Care must be taken regarding stack references, because COMPACT defines a separate stack segment and SMALL does not. For more information on 80386 microprocessor segment combining, see the binder chapter in the utilities user's guide.

end PL/M-386

13.4.2 Closed Subsystems

A closed subsystem differs from an open subsystem in two ways: it has a name and it consists of a specific list of modules. The compiler must know the name of the subsystem and the modules belonging to the subsystem in order to create a closed subsystem.

The need for a closed subsystem name is simply to differentiate a particular closed subsystem from another closed subsystem or from the open subsystem. This is done as follows: the name of the subsystem is added to the beginning of the segment names to create unique code and data segments.

For example, if a subsystem is named PHASE1, then the code sections from all modules belonging to the PHASE1 subsystem are combined into a single code segment called PHASE1_CODE32; similarly for COMPACT subsystems the data sections are combined into a single data segment called PHASE1_DATA. When using COMPACT, however, the stack sections are still combined into a segment called STACK because only one execution-time stack is usually necessary. Using SMALL all data and stack segments are combined in one segment called DATA, as usual.

A closed subsystem module list is needed for differentiation. For instance, if the compiler is not informed that module SCANNER belongs to subsystem PHASE1, then the compiler has no choice but to assume that module SCANNER belongs to the open subsystem.

Thus, every module in a program either is specified as part of a closed subsystem or, by default, becomes part of the open subsystem. A program can consist of only closed subsystems, or of both closed subsystems and the open subsystem, or of only the open subsystem (by default). There is only one open subsystem per program; all open subsystems are treated as one subsystem by the utility used to combine the program modules.

13.4.3 Communication between Subsystems

Within a subsystem there can be code and/or data items (procedures and variables) that must be known by other subsystems; that is, they are meant to be referenced from other subsystems. Such items are said to be exported. The export of a symbol is not directed at any one particular subsystem; it is directed at all subsystems in the program, including its own subsystem.

It is important to realize that the subsystem definitions are additions to normal intermodule PUBLIC/EXTERNAL definitions, not replacements.

For instance, module MOD1 belongs to subsystem SUB1 and makes a reference to symbol SYM2; SYM2 belongs to subsystem SUB2. SYM2 must be declared as EXTERNAL in MOD1, as usual, and must also be declared as PUBLIC and exported from SUB2. Using this information, the compiler generates an intersegment reference to SYM2.

13.5 Syntax

Defining subsystems means telling the compiler what extended segmentation model each subsystem uses, and which modules belong to each subsystem. In addition, it means telling the compiler which procedures and data are accessible from outside the subsystem.

Making everything available to all subsystems defeats the purpose of subsystems. For example, if a procedure is declared to be accessible from outside the subsystem, it is a far procedure. This means that all calls are far calls, even if the procedure is never actually accessed from outside its subsystem.

Each subsystem in a PL/M program has one extended segmentation model definition, which takes one of the following forms:

1. $ *model* (*subsystem-id* [*submodel*]

```
┌ ┌                   ┐ ┌ ┌                   ┐ ┐     ┐
│ │ HAS module-list   │ │ │ HAS module-list   │ │     │
│ {                   } │ ; {                 } │ ... │ )
│ │ EXPORTS public-list │ │ EXPORTS public-list │     │
└ └                   ┘ └ └                   ┘ ┘     ┘
```

2. $ *model* ([*submodel*]

```
{ HAS module-list     } [ { HAS module-list     } ]
{                     } [ ;                       ] )
{ EXPORTS public-list } [ { EXPORTS public-list } ]
```

3. $ *model* (*submodel*

```
┌ ┌                   ┐ ┌ ┌                   ┐ ┐     ┐
│ │ HAS module-list   │ │ │ HAS module-list   │ │     │
│ {                   } │ ; {                 } │ ... │ )
│ │ EXPORTS public-list │ │ EXPORTS public-list │     │
└ └                   ┘ └ └                   ┘ ┘     ┘
```

Where:

model	is SMALL, COMPACT, or LARGE and specifies the extended segmentation model for the subsystem. All modules in the subsystem must be compiled with the same extended segmentation model.
submodel	is -CONST IN CODE- or -CONST IN DATA- and specifies the placement of constants. -CONST IN CODE- corresponds to the ROM submodel; -CONST IN DATA- corresponds to the RAM submodel.

The default depends on the segmentation control and corresponds to the defaults of RAM/ROM for each model. The use of the RAM and ROM controls (see Chapter 11) can create conflicts when subsystems are defined. RAM is specified by -CONST IN DATA-; ROM is specified by -CONST IN CODE-.

subsystem-id — is any PL/M identifier that can be used as a module name, and specifies the name of the subsystem. This ID does not conflict with any IDs used within the program. A subsystem control without *subsystem-id* defines the open subsystem.

HAS *module-list* — is a list of module names, separated by commas, specifying the modules belonging to the subsystem. These module names must exactly match the module names from each source file comprising the subsystem. (A module name is the name of the outermost DO block of a source file.) A particular module name can appear in only one *module-list*. There are no default modules in the *module-list*. Any module for which a name does not appear in a *module-list* becomes part of the open subsystem.

EXPORTS *public-list* — is a list of procedure, variable, and constant IDs, specifying the code and data objects exported by the subsystem (i.e., accessible outside of the subsystem). Using a dollar sign ($) in a procedure name (within a subsystem definition) will cause an error. Any symbol in the exports list may be declared PUBLIC in at most one of the modules belonging to the subsystem, and should be declared EXTERNAL in all modules in and out of the subsystem that access the symbol.

A particular exported symbol can appear in only one *public-list*.

The *public-list* is exhaustive. Only the symbols in the *public-list* can be referenced from other subsystems. Symbols in the subsystem declared PUBLIC but not appearing in the *public-list* are accessible only from within the subsystem itself. Conversely, PUBLIC symbols that are not intended to be referenced from outside the subsystem should not appear in the *public-list*. These symbols are called domestic symbols.

In most applications of the subsystem controls, the HAS and EXPORTS lists will have several dozen entries apiece. To accommodate lists of this length, a subsystem control may be continued over more than one control line. (The continuation lines must be contiguous, and each must begin with a dollar sign ($) in the first column.)

Keep in mind that using a dollar sign in a procedure name within a subsystem definition will cause an error. Also, note that any number of HAS and EXPORTS lists can appear in a control, in any order. This enables formatting of the subsystem specification so it can be easily read and maintained.

Consider the following subsystem definition:

```
$COMPACT(SUB_INPUT -CONST IN CODE- HAS mod_1, mod_2, mod_3;
$               EXPORTS input)
$SMALL(SUB_PROCESS HAS mod_4, mod_5, mod_6, mod_7, mod_8)
$COMPACT(SUB_OUTPUT HAS mod_9, mod_10; EXPORTS format, output)
```

This sample program contains three subsystems: SUB_INPUT, SUB_PROCESS, and SUB_OUTPUT. SUB_INPUT and SUB_OUTPUT use the COMPACT extended segmentation model. SUB_PROCESS uses the SMALL extended segmentation model. Constants are stored with the code in SUB_INPUT. The SUB_INPUT subsystem contains the modules mod_1, mod_2, and mod_3, and exports one symbol, input. SUB_PROCESS contains modules 4 through 8. SUB_PROCESS contains the main program, as it must, since it is the only SMALL subsystem in the program (recall that when mixing SMALL with other models, the main program must be SMALL). For this reason it does not need to export any symbols. (A subsystem containing the main program can export symbols (for instance, global variables). But other subsystems MUST export at least one symbol, otherwise they are totally unaccessible to the main program, and therefore useless to the program of which they are a part.) SUB_OUTPUT supplies two symbols called format and output.

The preceding subsystem definition should appear in all 10 modules (mod_1 through mod_10), even though not all the exported symbols are used by all subsystems. It is recommended that the subsystem definition be kept in an INCLUDE file, then included in each module compiled. This avoids any problems in maintaining consistency between the subsystem definitions of all source modules.

Consider another example, this time containing an open subsystem. Start from an existing COMPACT program that does not use extended segmentation models, but whose code has grown too large. Assume that the following modules from the original program (ATTACH, OPEN, CLOSE, ERRORS, ALLOCATE, FREE) were compiled with the following segmentation control:

```
$COMPACT
```

If the modules ALLOCATE and FREE are factored out from the original program, creating SUBSYS1, the subsystem definition is as follows:

```
$COMPACT(SUBSYS1 HAS ALLOCATE, FREE)
```

Now, suppose that the modules remaining in the open subsystem reference entry points AllocBuff and FreeBuff in SUBSYS1. These must be exported from SUBSYS1 as follows:

```
$COMPACT(SUBSYS1 HAS ALLOCATE, FREE;
$                EXPORTS AllocBuff, FreeBuff)
```

or

```
$COMPACT(SUBSYS1 HAS ALLOCATE; EXPORTS AllocBuff;
$        HAS FREE; EXPORTS FreeBuff)
```

The second form illustrates how multiple HAS and EXPORTS lists can be used to document the items exported from each module.

If a routine in SUBSYS1 references the procedure FatalError in the module ERRORS, the definition of the open subsystem is as follows:

```
$COMPACT (EXPORTS FatalError)
```

No data structures need to be changed, because data reference values can be two bytes. All procedures except AllocBuff and FreeBuff use the short call and return mechanism.

13.5.1 Placement of Segmentation Controls

The segmentation controls have special restrictions associated with their placement. These rules are as follows:

- The segmentation controls are primary controls. They must appear before the DO statement of the module name.
- Only the definition of the open subsystem (with no submodel and no EXPORTS list) can be placed on the invocation line; definitions of all other subsystems must occur inside the source program.

The subsystem definitions for the entire program can be included in the compilation of each module using the INCLUDE control. The compiler extracts the information needed to correctly and efficiently compile each module's intrasubsystem and intersubsystem references.

13.6 Exporting Procedures

A symbol included in a subsystem's EXPORTS list must be declared PUBLIC in one of the modules in that subsystem. The symbol, called an exported symbol, can be referenced by modules in other subsystems. A PUBLIC symbol defined within a subsystem but not listed in its EXPORTS list is called a domestic symbol. It should be referenced only by modules within the same subsystem.

A procedure should be exported only if it must be referenced outside the defining subsystem, because accessing exported procedures will, in general, require more code and time than is required for domestic procedures.

Exported procedures have the following characteristics:

- The long form of call and return is used.
- The caller's DS register (and the ES register for the 80386 microprocessor) is saved and restored upon entry and exit.
- The DS register (and the ES register for the 80386 microprocessor) is loaded with the associated data segment upon entry.

NOTE

If a SMALL or MEDIUM module calls a procedure that is exported from a COMPACT or LARGE subsystem, the stack sections of the two will not be combined when the modules are combined because the segments containing them have different names (see Chapter 11). To get the proper stack size, the SEGSIZE control on the utility used to combine the program modules must be used to increase the size of the DATA segment. This segment must be increased by the sum of the stack requirements for both the SMALL or MEDIUM module and the subsystem.

The SMALL RAM segmentation control uses short pointers. Therefore, care must be taken when calling procedures that have pointer parameters and are exported from a SMALL subsystem. In these cases, the compiler always uses the value of the current DS register as the selector portion of the long pointer. This means that passing a pointer to any data items declared in the SMALL module will produce the proper result, but the following restrictions must be observed for the special cases:

1. If the actual parameter is the NIL pointer, DS:0 will be passed to the exported procedure. Consequently, the procedure executes differently if it is called from a SMALL module than if it had been called from a COMPACT, MEDIUM, or LARGE module. For example:

```
$COMPACT (FOO HAS N; EXPORTS FOO)
$SMALL
M: DO;
     DECLARE PTR POINTER;
     FOO: PROCEDURE (P) EXTERNAL;
          DECLARE P POINTER;
     END FOO;
     CALL FOO (NIL);              /* Wrong, will pass DS:0 */
     PTR = NIL;
     CALL FOO (PTR);              /* Wrong, will pass DS:0 */
END M;
$COMPACT (FOO HAS N; EXPORTS FOO)
    N: DO;
          FOO: PROCEDURE (P) PUBLIC;
                DECLARE P POINTER;
                DECLARE B BYTE;
                B = (P=NIL);/* Will assign FALSE (000H) to B
                                    if FOO is called from SMALL;
                                    Will assign TRUE (0FFH) to B
                                    otherwise */
          END FOO;
          CALL FOO (NIL);           /* Right, will pass 0:0 */
END N;
```

2. If the actual parameter is a pointer to a procedure, the compiler extends the short pointer with DS and then passes the value of DS:(offset of procedure) to the exported procedure. This situation should be avoided because the result of any reference through such a pointer is undefined. For example:

```
$COMPACT (FOO HAS N; EXPORTS FOO)
$SMALL
M: DO;
    DECLARE PTR POINTER;
    DECLARE TABLE(10) BYTE;
    FOO: PROCEDURE (P) EXTERNAL;
        DECLARE P POINTER;
    END FOO;
    BAZ: PROCEDURE;
       ...
    END BAZ;

    CALL FOO (@BAZ);                     /* Wrong, will pass */
                                         /* DS:offset-of-BAZ */

    PTR = @BAZ;
    CALL FOO (PTR);              /* Wrong, will pass DS:PTR */
    CALL FOO (@TABLE);    /* Right, will pass pointer to TABLE */

END M;
```

13.6.1 Large Matrix Example

The large REAL matrix example can now be fully developed (see Section 13.4). Recall that one module for each row is needed, with each module containing a 1-billion element REAL array. Running such an application is possible only on systems having virtual memory management for supporting such large data. The first module could be:

```
ROW0_MOD: DO;             /* ROW0_MOD is the module name */
DECLARE ROW0 (1000000000) REAL PUBLIC;
END ROW0_MOD;
```

The modules for ROW1 through ROW9 are similar. The subsystem definition at this point is:

```
$COMPACT ( ROW0_SYS HAS ROW0_MOD; EXPORTS ROW0 )
$COMPACT ( ROW1_SYS HAS ROW1_MOD; EXPORTS ROW1 )
$COMPACT ( ROW2_SYS HAS ROW2_MOD; EXPORTS ROW2 )
$COMPACT ( ROW3_SYS HAS ROW3_MOD; EXPORTS ROW3 )
$COMPACT ( ROW4_SYS HAS ROW4_MOD; EXPORTS ROW4 )
$COMPACT ( ROW5_SYS HAS ROW5_MOD; EXPORTS ROW5 )
$COMPACT ( ROW6_SYS HAS ROW6_MOD; EXPORTS ROW6 )
$COMPACT ( ROW7_SYS HAS ROW7_MOD; EXPORTS ROW7 )
$COMPACT ( ROW8_SYS HAS ROW8_MOD; EXPORTS ROW8 )
$COMPACT ( ROW9_SYS HAS ROW9_MOD; EXPORTS ROW9 )
```

Now define the program:

```
MATRIX_MOD: DO;
DECLARE ROW0 (1000000000) REAL EXTERNAL;
DECLARE ROW1 (1000000000) REAL EXTERNAL;
DECLARE ROW2 (1000000000) REAL EXTERNAL;
DECLARE ROW3 (1000000000) REAL EXTERNAL;
DECLARE ROW4 (1000000000) REAL EXTERNAL;
DECLARE ROW5 (1000000000) REAL EXTERNAL;
DECLARE ROW6 (1000000000) REAL EXTERNAL;
DECLARE ROW7 (1000000000) REAL EXTERNAL;
DECLARE ROW8 (1000000000) REAL EXTERNAL;
DECLARE ROW9 (1000000000) REAL EXTERNAL;

DECLARE ROW_POINTERS (10) POINTER INITIAL (
@ROW0, @ROW1, @ROW2, @ROW3, @ROW4,
@ROW5, @ROW6, @ROW7, @ROW8, @ROW9 );
RETRIEVEELEMENT: PROCEDURE (ROW,COL) REAL PUBLIC;
        DECLARE (ROW,COL) WORD;
        DECLARE ROWPTR POINTER,
    ROW_ARRAY BASED ROW_PTR (1) REAL;
        ROW_PTR = ROW_POINTERS (ROW);
        RETURN ROW_ARRAY (COL);
END RETRIEVE_ELEMENT;
STORE_ELEMENT: PROCEDURE (ROW,COL,VAL) PUBLIC;
        DECLARE (ROW,COL) WORD;
        DECLARE VAL REAL;
        DECLARE ROW_PTR POINTER,
ROW_ARRAY BASED ROW_PTR (1) REAL;
        ROW_PTR = ROW_POINTERS (ROW);
        ROW_ARRAY (COL) = VAL;
END STORE_ELEMENT;
                /* the matrix processing code inserted here */
END MATRIX_MOD;
```

Now assume that other modules will be added to this program later. In this case, it is better to put MATRIX_MOD and these other modules in the COMPACT OPEN subsystem. This way modules can freely be added or deleted without having to redefine the overall subsystem structure. Also assume the need to calculate sines and cosines of various matrix elements. The functions SINE and COSINE are supplied in an external math package. The only thing known about this package is that all its routines require long calls.

The final subsystem definition is now:

```
$LARGE ( EXPORTS SINE, COSINE )
$COMPACT ( ROW0_SYS HAS ROW0_MOD; EXPORTS ROW0 )
$COMPACT ( ROW1_SYS HAS ROW1_MOD; EXPORTS ROW1 )
$COMPACT ( ROW2_SYS HAS ROW2_MOD; EXPORTS ROW2 )
$COMPACT ( ROW3_SYS HAS ROW3_MOD; EXPORTS ROW3 )
$COMPACT ( ROW4_SYS HAS ROW4_MOD; EXPORTS ROW4 )
$COMPACT ( ROW5_SYS HAS ROW5_MOD; EXPORTS ROW5 )
$COMPACT ( ROW6_SYS HAS ROW6_MOD; EXPORTS ROW6 )
$COMPACT ( ROW7_SYS HAS ROW7_MOD; EXPORTS ROW7 )
$COMPACT ( ROW8_SYS HAS ROW8_MOD; EXPORTS ROW8 )
$COMPACT ( ROW9_SYS HAS ROW9_MOD; EXPORTS ROW9 )
```

The COMPACT control should appear in the invocation line. The first control line indicates that the symbols SINE and COSINE require long references and belong to some unknown subsystem. The next ten lines define the ten closed subsystems, each containing a row of the matrix. The COMPACT control is specified on the invocation line when compiling MATRIX_MOD (and when compiling any other module in the program except the ROW modules).

Because every module in this program should be compiled with the same subsystem definitions, it is convenient to put these control lines in an INCLUDE file. If any changes to the subsystem definitions are made later, only one file needs to be updated.

Error and Warning Messages 14

Tabs for
452161-001

14 ERROR AND WARNING MESSAGES CONTENTS

Page

14 ERROR AND WARNING MESSAGES

The compiler may issue five varieties of error and warning messages:

- PL/M program error messages
- Fatal command tail and control error messages
- Fatal input/output error messages
- Fatal insufficient memory error messages
- Fatal compiler failure error messages
- Insufficient memory warning messages

The source errors are reported in the program listing; the fatal errors are reported on the console device.

14.1 PL/M Program Error and Warning Messages

Nearly all of the source PL/M program error messages are interspersed in the listing at the point of error and follow the general format:

```
***ERROR mmm IN nnn (LINE ppp), NEAR 'aaa', message
```

or:

```
***WARNING mmm IN nnn (LINE ppp), NEAR 'aaa', message
```

Where:

mmm	is the error number from the following list of source error messages.
nnn	is the source statement number where the error occurs.
ppp	is the actual source line number where the error occurs.
aaa	is the source text near where the error is detected.
message	is the error explanation from the following list of source error messages.

The following source error messages may be encountered.

```
***ERROR 1 INVALID CONTROL
```

An unrecognized control in the control line; for example:

```
$NXCODE;                          /* probably intended NOCODE */
```

```
***ERROR 2 PRIMARY CONTROL FOLLOWS NON-CONTROL LINE
```

Primary controls can appear as control lines in the source program, but they must come first. No other statements can precede them.

```
***ERROR 3 MISSING CONTROL PARAMETER
```

Certain controls (e.g., INCLUDE), require the specification of a parameter.

```
***ERROR 4 INVALID CONTROL PARAMETER
```

Examples are an illegal pathname for a control such as OBJECT or a string where a number is expected.

```
***ERROR 5 INVALID CONTROL FORMAT
```

See Chapter 11 for correct formatting of control lines. Following is an example that could cause this error:

```
$LIST (MYPROG.LST);
```

This error could occur because no *pathname* is expected on this control. It could also be caused if an INCLUDE was followed by another control on the same line.

```
***ERROR 6
```

Not used.

```
***ERROR 7 INVALID PATHNAME
```

The pathname for a file has been incorrectly specified; see the host-system operating instructions.

```
***WARNING 8 ILLEGAL PAGELENGTH, IGNORED
```

The pagelength specified is less than 5 or greater than 255; the default is 60.

```
***ERROR 9 ILLEGAL PAGEWIDTH, IGNORED
```

The pagewidth specified is less than 60 or more than 132; the default is 120.

```
***WARNING 10 RESPECIFIED PRIMARY CONTROL, IGNORED
```

Primary controls can be specified only once and cannot alter a previous setting.

```
***ERROR 11 MISPLACED ELSE OR ELSEIF CONTROL
```

ELSE or ELSEIF control occurred without being preceded by a corresponding IF control.

```
***ERROR 12 MISPLACED ENDIF CONTROL
```
ENDIF control occurred without being preceded by a corresponding IF control.

```
***ERROR 13 MISSING ENDIF CONTROL
```
End of source file without an ENDIF control to match a previous IF.

```
***ERROR 14 NAME TOO LONG(31), TRUNCATED
```
Switch variable name in IF, ELSE, SET, or RESET statement is too long.

```
***ERROR 15 MISSING OPERATOR
```
Two operands in an expression must be separated by an arithmetic, logical, or relational operator.

```
***WARNING 16 INVALID CONSTANT, ZERO ASSUMED
```
The constant specified by SET, IF, or ELSEIF is invalid.

```
***ERROR 17 INVALID OPERAND
```
SET, RESET, IF, or ELSEIF were used in an invalid position.

```
***WARNING 18 PARENTHESES IGNORED WITHIN CONDITIONAL
  COMPILATION CONDITION
```
Parentheses within conditional compilation conditions are ignored and the expression is evaluated according to the regular precedence rules.

```
***ERROR 19 LIMIT EXCEEDED: SAVE NESTING
```
See Appendix B for the correct limit.

```
***ERROR 20 LIMIT EXCEEDED: INCLUDE NESTING
```
For example, a file named A is included, which includes a file named B, and so on. This error will occur when the limit is exceeded. See Appendix B for the correct limit.

```
***ERROR 21 MISPLACED RESTORE CONTROL
```
RESTORE can work only if there has been a prior SAVE.

```
***ERROR 22 UNEXPECTED END OF CONTROL
```
A segmentation control was expecting a continuation line or a right parenthesis.

```
***ERROR 23 SYMBOL EXISTS IN MORE THAN ONE HAS LIST
```
A module name can occur in only one HAS list.

```
***ERROR 24 SUBSYSTEM ALREADY DEFINED
```
The subsystem name has already been defined.

***ERROR 25 CONFLICTING SEGMENTATION CONTROLS

More than one segmentation control affecting the module being compiled was encountered. One common cause is specifying both -CONST IN CODE- and ROM in a module with a subsystem definition.

***ERROR 26 ILLEGAL PL/M IDENTIFIER

Identifier does not meet the rules for PL/M identifiers. Identifiers can be up to 31 alphanumeric characters or the underscore. However, the first character must be alphabetic or the underscore.

***ERROR 27 PREDEFINED SWITCHES ARE NOT VALID BEFORE MODULE NAME

Predefined switches can be used only after the first DO statement.

Error 27 is not used by PL/M-386.

***WARNING 28 INVALID PL/M CHARACTER, IGNORED

Look near the text flagged for an invalid character, or one that is inappropriate in context. Delete it or retype the statement.

***WARNING 29 UNPRINTABLE CHARACTER, IGNORED

Retype the line in question using valid characters.

***ERROR 30 STRING TOO LONG, TRUNCATED

Match the intended variable type with the length of the flagged item to obtain the correct maximum length. See Appendix B for the correct limit.

***ERROR 31 ILLEGAL CONSTANT TYPE

A constant contains illegal characters. This might reflect missing operators (e.g., A = 4T instead of A = 4 + T).

***ERROR 32 INVALID CHARACTER IN CONSTANT

For example, 107B and 0ABCD will cause this error because neither can be valid in any PL/M interpretation; 7 is not a binary numeral, B cannot occur in decimal or octal, and neither string ends in H.

***ERROR 33 RECURSIVE MACRO EXPANSION

Following is an example causing this error:

```
DECLARE A LITERALLY 'B';
DECLARE B LITERALLY 'A';
 .
 .
 .
B=4;                          /* error discovered here */
```

LITERALLYs cannot be declared circularly (i.e., solely in terms of each other).

***ERROR 34 LIMIT EXCEEDED: MACRO NESTING (5)

This error occurs when too many DECLARE statements refer back through each other to the one that actually supplies a type. See Appendix B for the correct limit. For example:

```
DECLARE A LITERALLY 'B';
DECLARE B LITERALLY 'C';
   . . . .
   . . . .
DECLARE Y LITERALLY 'Z';
DECLARE Z BYTE INITIAL(77);
   .
   .
A=7;                          /* error discovered here */
```

***ERROR 35 LIMIT EXCEEDED: SOURCE LINE LENGTH (128)

See Appendix B for the correct limit.

***ERROR 36

Not used.

***ERROR 37 INVALID REAL CONSTANT

***WARNING 38 REAL CONSTANT UNDERFLOW

An underflow occurred when conversion into floating-point was attempted.

***WARNING 39 REAL CONSTANT OVERFLOW

An overflow occurred when conversion into floating-point was attempted.

***ERROR 40 NULL STRING NOT ALLOWED

Strings of length zero are not supported.

***ERROR 41 DELETED: "*<tokens>*"

The compiler deleted tokens while attempting to recover from a syntax error.

`***ERROR 42 INSERTED:"<tokens>"`

The compiler inserted tokens while attempting to recover from a syntax error.

`***ERROR 43 STATEMENTS FOLLOW MODULE END`

Statements follow the logical end-of-module.

`***ERROR 44 CONSTANT TOO LARGE`

A constant value (e.g., 999,999,999,999) is too large for the compiler.

`***WARNING 45 MISMATCHED BLOCK IDENTIFIER`

If a label is supplied in an END statement, the label must match the first unmatched DO statement above the END. Sometimes the error involves a module name confused with a procedure name.

`***ERROR 46 DUPLICATE PROCEDURE NAME`

Procedure names must be unique.

`***ERROR 47 LIMIT EXCEEDED: PROCEDURES`

Too many procedures in this module. Break it into smaller modules. See Appendix B for the correct limit.

`***ERROR 48 DUPLICATE PARAMETER NAME`

A parameter must be declared exactly once. This message indicates that the flagged parameter already has a definition at this block level, as in:

```
YAR: PROCEDURE (YAR77, YAR78);
DECLARE YAR77 BYTE;
DECLARE YAR77 BYTE;
```

Perhaps a different spelling was intended.

`***ERROR 49 NOT AT MODULE LEVEL`

The flagged attribute or initialization can be valid only at the module level, not in a procedure.

`***ERROR 50 DUPLICATE ATTRIBUTE`

Attributes should be specified only once. This message means the compiler has found a declaration like:

```
DECLARE B BYTE EXTERNAL EXTERNAL;
```

`***ERROR 51 MISSING OR ILLEGAL INTERRUPT VALUE`

Interrupt numbers must be whole-number constants between 0 and 255. Thus -7 or 272 would be invalid.

`***ERROR 52 INTERRUPT WITH PARAMETERS`

No parameters can be used in interrupt procedures.

`***ERROR 53 INTERRUPT WITH TYPED PROCEDURE`
Interrupt procedures must be untyped.

`***ERROR 54 INVALID DIMENSION`

`***ERROR 55 LIMIT EXCEEDED: NESTED STRUCTURES`
See Appendix B for the correct limit.

`***ERROR 56 STAR DIMENSION WITH STRUCTURE MEMBER`
Star dimension (*) must not be used with structures. The dimensions for an array that is a structure member must be specified explicitly.

`***ERROR 57 CONFLICT WITH PARAMETER`
Object cannot be a parameter.

`***ERROR 58 DUPLICATE DECLARATION`
The flagged item already has a definition declared at this block level.

`***ERROR 59 ILLEGAL PARAMETER TYPE`
Parameters cannot be declared of type structure or array.

`***ERROR 60 DUPLICATE LABEL`
Each label must be unique within its block or scope. Otherwise, GOTOs and CALLs would have ambiguous targets.

`***ERROR 61 DUPLICATE MEMBER NAME`
Member has been declared twice in the same structure. For example, in:

```
DECLARE AIR STRUCTURE (F4 BYTE, F4 BYTE);
```

subsequent references to AIR.F4 would be ambiguous.

`***ERROR 62 UNDECLARED PARAMETER`
A parameter named in the procedure statement was not defined in the body of the procedure.

`***ERROR 63 CONFLICTING ATTRIBUTES`
A variable has been declared with inconsistent attributes (e.g., PUBLIC or EXTERNAL, DATA or INITIAL, AT or BASED).

`***ERROR 64 LIMIT EXCEEDED: DO BLOCKS`
See Appendix B for the correct limit.

`***ERROR 65 ILLEGAL PARAMETER ATTRIBUTE`
Certain attributes cannot be used to declare a parameter (e.g., PUBLIC, EXTERNAL, DATA, INITIAL, AT, or BASED).

```
***ERROR 66 UNDEFINED BASE
```
A variable was declared BASED using an undeclared identifier.

```
***ERROR 67 INVALID TYPE OR ATTRIBUTE FOR BASE
```
A base must be a non-subscripted scalar of type ADDRESS, POINTER, WORD, SELECTOR, or OFFSET (PL/M-386 only).

```
***ERROR 68 MISPLACED DECLARATION
```
Declarations and procedures can be interspersed, but not declarations and executable statements.

```
***ERROR 69 INVALID BASE WITH LABEL OR MACRO
```
BASED cannot be used with LABEL or LITERALLY types.

```
***ERROR 70 INVALID DIMENSION WITH LABEL OR MACRO
```
LABEL or LITERALLY cannot be dimensioned.

```
***ERROR 71 INITIALIZATION LIST REQUIRED
```
A list of initial values is required if the INITIAL attribute, the non-external * dimension form, or the non-external DATA attribute is used.

```
***ERROR 72 BASED CONFLICTS WITH ATTRIBUTES
```
Examples of attributes conflicting with base include AT, DATA, INITIAL, PUBLIC, and EXTERNAL.

```
***ERROR 73 DATA OR EXECUTABLE STATEMENTS IN
   EXTERNAL
```
An EXTERNAL procedure, being defined elsewhere, cannot contain executable statements or data declarations for variables that are not formal parameters.

```
***ERROR 74 MISSING RETURN FOR TYPED PROCEDURE
```
A typed procedure must return a value; thus, it must include a RETURN statement.

```
***ERROR 75 INVALID NESTED REENTRANT PROCEDURE
```
Reentrant procedures cannot contain procedures.

```
***ERROR 76 LIMIT EXCEEDED: FACTORED LIST
```
Too many variables were named in a factored declaration. Break it into several declarations. See Appendix B for the correct value.

```
***ERROR 77 LIMIT EXCEEDED: STRUCTURE MEMBERS
```
See Appendix B for the correct value.

```
***ERROR 78 MISSING PROCEDURE NAME
```
Every procedure must have a name.

***ERROR 79 MULTIPLE PROCEDURE LABELS
Procedures must have only one name.

***ERROR 80 DECLARATIONS MAY NOT BE LABELED
Labels cannot be used on declaration statements.

***ERROR 81 STAR DIM WITH FACTORED LIST NOT ALLOWED
Separate the array declarations giving the data initializations for each array separately, or explicitly state the dimensions of the factored array declarations as in the following examples:

```
DECLARE (A,B) (*) BYTE DATA ('abcd', 'xywz');   /* is illegal */
DECLARE (A) (*) BYTE DATA ('abcd');               /* is legal */
DECLARE (B)(*) BYTE DATA ('xywz');                /* is legal */
```

or

```
DECLARE (A,B) (4) BYTE DATA ('abcd', 'xywz');     /* is legal */
```

***ERROR 82 SIZE EXCEEDS *nn* BYTES
Storage for the declared item exceeds the maximum storage for the microprocessor. For the 8086 and the 80286 microprocessors, *nn* is 64K. For the 80386 microprocessor, *nn* is 4G.

***WARNING 83 PROCEDURE CONTAINS NO EXECUTABLE STATEMENTS
This procedure does nothing, but executes successfully.

***ERROR 84
Not used.

***ERROR 85 INITIAL USED WITH ROM OPTION
Variables declared with INITIAL are not initialized until load-time. Thus, if the program is in ROM, these initializations will never occur.

***ERROR 86 LIMIT EXCEEDED: NUMBER OF PARAMETERS
The procedure declaration includes too many parameters. See Appendix B for the correct limit.

***ERROR 87
Not used.

***ERROR 88 LIMIT EXCEEDED: PROGRAM TOO COMPLEX
The program has too many complex expressions, cases, or procedures. Break it into smaller procedures.

`***ERROR 89 COMPILER ERROR: BAD ERROR RECOVERY`

An unrecoverable error occurred. Trying a different copy of the compiler on a different drive might reveal that the first copy had been damaged. Contact your Intel representative.

`***ERROR 90 COMPILER ERROR: MULTIPLE PARSE ARGS`

See source error message number 89.

`***ERROR 91 LIMIT EXCEEDED: PROGRAM TOO COMPLEX`

The program has too many complex expressions, cases, or procedures. Break it into smaller procedures.

`***ERROR 92 COMPILER ERROR: PARSE ARG STACK UNDERFLOW`

See source error message number 89.

`***ERROR 93 LIMIT EXCEEDED: PROGRAM TOO COMPLEX`

The program has too many complex expressions, cases, or procedures. Break it into smaller modules.

`***ERROR 94 COMPILER ERROR: PARSE STACK UNDERFLOW`

See source error message number 89.

`***ERROR 95 COMPILER ERROR: PARSE BUFFER OVERFLOW`

See source error message number 89.

`***ERROR 96 LIMIT EXCEEDED: BLOCK NESTING`

The program has too many nested DO blocks. Break it into smaller procedures. See Appendix B for the correct limit.

`***ERROR 97 COMPILER ERROR: SCOPE STACK UNDERFLOW`

See source error message number 89.

`***ERROR 98 LIMIT EXCEEDED: STATEMENT TOO COMPLEX`

The statement is too large for the compiler. Break it into several smaller statements.

`***ERROR 99 COMPILER ERROR: SEMANTIC UNDERFLOW`

See source error message number 89.

`***ERROR 100 STRING CONSTANT TOO LONG`

String constants used as scalars have a maximum of four characters.

`***ERROR 101 UNSUBSCRIPTED ARRAY`

The array reference requires a subscript.

```
***WARNING 102 UNQUALIFIED STRUCTURE
```
This statement is ambiguous as to which structure or member it references.

```
***ERROR 103 NOT AN ARRAY
```
Subscripts are permitted only on identifiers declared as arrays. Check spelling consistency.

```
***ERROR 104 MULTIPLE SUBSCRIPTS
```
PL/M has only single dimension arrays. Therefore, only one subscript is permitted in an array reference. For example, for any array TING references of the form TING(2,4) or TING(3,7,9,6) are invalid.

```
***ERROR 105 NOT A STRUCTURE
```
For example, a reference of the form GNU.F1, where GNU was not declared a structure.

```
***ERROR 106 UNDEFINED IDENTIFIER
```
Every identifier must be declared.

```
***ERROR 107 UNDEFINED MEMBER
```
For example, KAPI.HORN, where KAPI is a valid, declared structure but HORN is an undeclared member of the structure.

```
***ERROR 108 ILLEGAL ITERATIVE DO INDEX TYPE
```
Only expressions of type BYTE, WORD, and INTEGER can be used.

```
***ERROR 109 UNDEFINED OR NOT A LABEL
```
The identifier following GOTO must be a label; the flagged item was declared otherwise, or the identifier was declared as a label but was not defined.

```
***ERROR 110 MISSING RETURN VALUE
```
A typed procedure must return a value that is specified by its RETURN statement.

```
***ERROR 111 INVALID RETURN WITH UNTYPED PROCEDURE
```
An untyped procedure does not return a value; thus, its RETURN statement cannot specify one.

```
***ERROR 112 INVALID INDIRECT TYPE
```
Only WORD or POINTER scalars can be used for indirect calls. This excludes WORD or POINTER expressions, BYTE, DWORD, INTEGER, or REAL scalars, all structures, and all arrays.

```
***ERROR 113 INVALID PARAMETER COUNT
```
The number of actual parameters supplied in a CALL must be equal to the number of formal parameters declared in the procedure.

```
***ERROR 114 QUALIFIED PROCEDURE NAME
```
Procedure names cannot be qualified.

```
***ERROR 115 INVALID FUNCTION REFERENCE
```
Typed procedures can be invoked only by use in an expression, not by a CALL.

```
***ERROR 116 INVALID CASE EXPRESSION TYPE
```
Case expressions must be of type BYTE, WORD, or INTEGER.

For PL/M-86 and PL/M-286, error 117 displays the following message:

```
***ERROR 117 LIMIT EXCEEDED: NUMBER OF CASES
```
See Appendix B for the correct limit.

For PL/M-386, error 117 displays the following message:

```
***ERROR 117 LIMIT EXCEEDED: NUMBER OF ACTIVE CASES
```
Reduce the number of cases in this case statement; the maximum number has been exceeded.

```
***ERROR 118 TYPE CONFLICT
```
An example of type conflict is WORD and REAL mixed in a reference.

```
***ERROR 119 INVALID BUILT-IN REFERENCE
```
Built-in reference was qualified with a member name, or OUTPUT/OUTWORD did not appear on the left side of an assignment.

```
***ERROR 120 INVALID PROCEDURE REFERENCE
```
Untyped procedures must be invoked by a CALL statement; references to such procedures are not permitted in expressions.

```
***ERROR 121 INVALID LEFT-HAND SIDE OF ASSIGNMENT
```
The left-hand side of the assignment must be a scalar variable. For example, PROCEDURE = 4 or INWORD(7) = 9.

```
***ERROR 122 INVALID REFERENCE
```
Invalid label reference.

```
***ERROR 123 USE OF "." MAY BE UNSAFE
```
The "dot" operator does not always produce correct results in a PL/M program that contains more than one data segment or more than one code segment.

```
***ERROR 124 PROCEDURE NAME REQUIRED
```
Procedure name is required for SET$INTERRUPT and INTERRUPT$PTR built-ins.

For PL/M-286 and PL/M-386, error 124 is not used.

```
***ERROR 125 PROCEDURE NAME ONLY
```
Parameters are not allowed on the procedure name in SET$INTERRUPT and INTERRUPT$PTR.

For PL/M-286 and PL/M-386, error 125 is not used.

For PL/M-86, error 126 displays the following message:
```
***ERROR 126 BAD INTERRUPT NUMBER (<=255)
```
For PL/M-386, error 126 displays the following message:
```
***ERROR 126 BAD INTERRUPT NUMBER
```
Interrupt numbers in a CAUSE$INTERRUPT statement must be whole-number constants in the range (0 - 255).

For PL/M-286, error 126 is not used.

```
***ERROR 127 CONSTANT ONLY
```
In this instance, a constant is required.

```
***ERROR 128 ARRAY REQUIRED
```
Some built-ins need an array name as a parameter.

```
***ERROR 129 INTERRUPT PROCEDURE REQUIRED
```
The name declared in a SET$INTERRUPT procedure or INTERRUPT$PTR function must be a previously declared procedure.

For PL/M-286 and PL/M-386, error 129 is not used.

```
***ERROR 130 INVALID RESTRICTED OPERAND
```
Illegal use of a dot operator.

```
***ERROR 131 INVALID RESTRICTED OPERATOR
```
Only + and − can be used in restricted expressions.

```
***ERROR 132
```
Not used.

```
***ERROR 133 REFERENCE REQUIRED
```
A variable reference is required for LENGTH, LAST, and SIZE.

```
***ERROR 134 VARIABLE REQUIRED
```
The operand to LENGTH, LAST, and SIZE must be a variable.

```
***ERROR 135 VALUE TOO LARGE
```
A value is too large for its contextually determined type.

```
***ERROR 136 ABSOLUTE POINTER WITH SHORT POINTERS
```
Two possible causes in the SMALL (RAM) case: pointer variables cannot be initialized with or assigned whole number constants, or the @ operator cannot be used with a variable that was located at an absolute address that was specified by a whole number constant.

```
***ERROR 137 INVALID RESTRICTED EXPRESSION
```
Only addresses or constant types can be used in restricted expressions.

```
***ERROR 138 PUBLIC AT EXTERNAL
```
PUBLIC declarations must be fully defined within the procedure. For example:

```
DECLARE DARTH BYTE EXTERNAL;
DECLARE VADER BYTE PUBLIC AT (.DARTH);
```

is illegal.

```
***ERROR 139 PUBLIC AT ABSOLUTE
```
Absolute locations for PUBLICS are supported only under the LARGE option.

For PL/M-286 and PL/M-386, error 139 is not used.

```
***ERROR 140 PUBLIC AT MEMORY
```
PUBLIC at @ MEMORY is not supported by SMALL.

For PL/M-286 and PL/M-386, error 140 is not used.

```
***ERROR 141 AT BASED VARIABLE
```
Based variables cannot be used in AT clauses.

```
***ERROR 142 ILLEGAL FORWARD REFERENCE
```
An AT expression cannot have a forward reference. Any location reference in the AT expression must refer to previously declared variables.

```
***ERROR 143 VARIABLE TYPE REQUIRED IN AN AT EXPRESSION
```
The AT expression must be a variable name. For example:

```
DECLARE B BYTE AT (.proc_name);
```

is illegal.

```
***ERROR 144 LIMIT EXCEEDED: DATA OR STACK SEGMENT
 TOO LARGE
```
See Appendix B for the correct limit.

```
***ERROR 145 LIMIT EXCEEDED: CODE OR CONST SEGMENT
 TOO LARGE
```
See Appendix B for the correct limit.

```
***ERROR 146 LIMIT EXCEEDED: NUMBER OF EXTERNALS
```
See Appendix B for the correct limit.

```
***ERROR 147 LABEL NOT AT LOCAL OR MODULE LEVEL
```
Label was not used correctly.

```
***ERROR 148 INITIALIZING MORE SPACE THAN DECLARED
```
The number of initialization values exceeds the number of declared elements.

```
***ERROR 149 ILLEGAL MODULE NAME REFERENCE
```
Module names cannot be referenced.

```
***WARNING 150 USE OF "." WITH FAR PROCEDURE
```
A subsequent indirect call made through the respective address/pointer generates the wrong type of call.

```
***WARNING 151 USE OF "@" WITH NEAR PROCEDURE
```
See source error message number 150.

```
***ERROR 152 INVALID "." OR "@" OPERAND
```
Must be used with a variable, procedure, or constant list.

```
***ERROR 153 INVALID RETURN IN MAIN PROGRAM
```
A main program must have no returns.

```
***ERROR 154 STAR DIMENSIONED VARIABLE WITH LENGTH,
   SIZE OR LAST
```
The LENGTH, LAST, and SIZE built-in functions cannot be used with variables declared with the implicit dimension specifier (*) and the EXTERNAL attribute.

```
***ERROR 155 SYMBOL EXPORTED FROM ANOTHER SUBSYSTEM
```
A PUBLIC symbol in this module is also exported by another subsystem.

```
***ERROR 156 LONG POINTER REQUIRED FOR THIS
   CONSTRUCT
```
A model with long pointers is required.

```
***ERROR 157
```
Not used.

```
***ERROR 158 INITIALIZATION CONFLICTS WITH
   ATTRIBUTES
```
An external variable cannot be initialized.

```
***ERROR 159 ILLEGAL INTERRUPT PROCEDURE REFERENCE
```
An interrupt procedure cannot be invoked with the CALL statement.

For PL/M-86, error 159 is not used.

```
***ERROR 160 INTERRUPT PROCEDURES MUST BE PUBLIC
```
An interrupt procedure must also be given the PUBLIC attribute.

```
***ERROR 161 ILLEGAL ABSOLUTE POINTER OR SELECTOR
```
Constants cannot be assigned to POINTERs or SELECTORs, nor used to initialize them. POINTERs and SELECTORs also cannot be passed as actual parameters.

For PL/M-86, error 161 is not used.

```
***ERROR 162 LIMIT EXCEEDED: STATEMENT TOO COMPLEX
```
The statement is too large for the compiler. Break it into several smaller statements.

```
***WARNING 162 LIMIT EXCEEDED: PROGRAM COMPLEXITY
```
Too many complex expressions, cases, etc. Break it into smaller procedures.

For PL/M-86 and PL/M-286, error 162 is not used.

```
***ERROR 163 COMPILER ERROR: SEMANTIC UNDERFLOW
```
See source error message number 89.

```
***ERROR 164 COMPILER ERROR: INVALID NODE
```
See source error message number 89.

```
***ERROR 165: 286 INTERFACE OBJECT NOT EXTERNAL
```
If the *machine* parameter is 286, all identifiers in the *id* list must be declared EXTERNAL.

For PL/M-286, error 165 is not used.

```
***ERROR 166 COMPILER ERROR: INVALID TREE
```
See source error message number 89.

```
***ERROR 167 COMPILER ERROR: SCOPE STACK UNDERFLOW
```
See source error message number 89.

For PL/M-86 and PL/M-286, error 168 displays the following message:
```
***ERROR 168 LIMIT EXCEEDED: BLOCK NESTING
```
See Appendix B for the correct limit.

For PL/M-386, error 168 displays the following message:

```
***ERROR 168 LIMIT EXCEEDED: PROGRAM COMPLEXITY
```

The program has too many complex expressions, cases, or procedures. Break it into smaller procedures.

```
***ERROR 169 COMPILER ERROR: INVALID RECORD
```

See source error message number 89.

For PL/M-86 and PL/M-286, error 169 is not used.

```
***ERROR 170 INVALID DO CASE BLOCK, AT LEAST ONE
   CASE REQUIRED
```

The DO CASE block is described in Section 6.1.

```
***ERROR 171 LIMIT EXCEEDED: NUMBER OF CASES
```

See Appendix B for the correct limit.

```
***ERROR 172 LIMIT EXCEEDED: NESTING OF TYPED
   PROCEDURE CALLS
```

See Appendix B for the correct limit.

```
***ERROR 173 LIMIT EXCEEDED: NUMBER OF ACTIVE
   PROCEDURES AND DO CASE GROUPS
```

See Appendix B for the correct limit.

```
***ERROR 174 ILLEGAL NESTING OF BLOCKS, ENDS NOT
   BALANCED
```

For every DO, an END is needed.

```
***ERROR 175 COMPILER ERROR: INVALID OPERATION
```

See source error message number 89.

```
***ERROR 176 LIMIT EXCEEDED: REAL EXPRESSION
   COMPLEXITY
```

The REAL stack has eight registers. Heavily nested use of REAL functions with REAL expressions as parameters can get excessively complex. See Appendix F.

```
***ERROR 177 COMPILER ERROR: REAL STACK UNDERFLOW
```

See source error message numbers 89 and 176.

```
***ERROR 178 LIMIT EXCEEDED: BASIC BLOCK
   COMPLEXITY
```

There is a very long list of statements without labels, CASEs, IFs, GOTOs, and/ or RETURNs. Either break the procedure into several smaller procedures, or add labels to some of the statements.

```
***ERROR 179 LIMIT EXCEEDED: STATEMENT SIZE
```
The statement is too large for the compiler. Break it into several smaller statements.

```
***ERROR 199 LIMIT EXCEEDED: PROCEDURE COMPLEXITY
   FOR OPTIMIZE(2)
```
The combined complexity of statements, user labels, and compiler-generated labels is too great. Simplify as much as possible, perhaps breaking the procedure into several smaller procedures.

```
***ERROR 200 ILLEGAL INITIALIZATION OF MORE SPACE
   THAN DECLARED
```
The number of initialization values exceeds the number of declared elements.

```
***ERROR 201 INVALID LABEL: UNDEFINED
```
No definition for this label was found.

```
***ERROR 202 LIMIT EXCEEDED: NUMBER OF EXTERNAL ITEMS
```
See Appendix B for the correct limit.

For PL/M-86 and PL/M-286, error 202 is not used.

```
***ERROR 203 COMPILER ERROR: BAD LABEL ADDRESS
```
See source error message number 89.

```
***ERROR 204 LIMIT EXCEEDED: CODE SEGMENT SIZE
```
See Appendix B for the correct limit.

```
***ERROR 205 COMPILER ERROR: BAD CODE GENERATED
```
See source error message number 89.

```
***ERROR 206 LIMIT EXCEEDED: DATA SEGMENT SIZE
```
See Appendix B for the correct limit.

```
***ERROR 207 ATTEMPT TO USE 0 AS DIVISOR IN
   DIVISION/MODULO
```
Zero cannot be used as a divisor in division/modulo; use 1. This error appears at the end as a semantic error.

```
***ERROR 210 COMPILER ERROR: OBJECT MODULE
   GENERATION ERROR
```
See source error message number 89.

For PL/M-86, error 210 is not used.

***ERROR 211 COMPILER ERROR: DEBUG SEGMENT SIZE OVERFLOW

See source error message number 89.

For PL/M-86, error 211 is not used.

***ERROR 212 COMPILER ERROR: ILLEGAL FIXUP

See source error message number 89.

For PL/M-86, error 212 is not used.

***ERROR 230 COMPILER ERROR: INVALID INTERNAL TYPE

See source error message number 89.

***ERROR 240 NO DEBUG INFORMATION FOR VARIABLES BASED ON BASED VARIABLES

Debug information is available only for the first level of indirection.

***ERROR 241 ILLEGAL TYPE CASTING

For example:

```
pt2=pointer(real_value)
```

is illegal.

***ERROR 242 TRUNCATION OF *n* BIT OFFSET

OFFSET was assigned to a variable with a size less than 32 bits; the assigned value may not be a valid OFFSET. For the 8086 and 80286 microprocessors, *n* is 16. For the 80386 microprocessor, *n* is 32.

For PL/M-86 and PL/M-286, error 243 displays the following message:

***ERROR 243 ILLEGAL INITIALIZATION WITH 0 CONSTANT: USE NIL

Use NIL to initialize POINTER to zero.

For PL/M-386, error 243 displays the following message:

***ERROR 243 286 INTERFACE OBJECT NOT EXTERNAL

If the *machine* parameter is 286, all identifiers in the *id* list must be declared EXTERNAL.

***ERROR 244 SYMBOL REPEATED IN INTERFACE SPECIFICATION

Symbols can only be used once in an INTERFACE control (i.e., a symbol cannot be repeated in the INTERFACE control).

For PL/M-86 and PL/M-286, error 244 is not used.

`***ERROR 245 AT VARIABLE IN DIFFERENT SEGMENT`

A variable cannot be declared using both the DATA attribute and the AT attribute when using the ROM option. DATA should be in CODE segments and INITIAL should be in DATA segments.

For PL/M-386, error 245 is not used.

`***WARNING 247 INDIRECT CALL THROUGH 16 BIT VARIABLE`

An indirect call through a 16-bit variable is not recommended because a 16-bit variable can address only the first 64K of a segment.

For PL/M-86 and PL/M-286, warning 247 is not used.

`***WARNING 248 BASE TYPE HAS ONLY 16 BITS OFFSET`

Use of a 16-bit base specifier is not recommended because it can address only the first 64K of a segment.

For PL/M-86 and PL/M-286, warning 248 is not used.

`***ERROR 251 COMPILER ERROR: INVALID OBJECT`

See source error message number 89.

`***ERROR 252 COMPILER ERROR: SELF NAME LINK`

See source error message number 89.

`***ERROR 253 COMPILER ERROR: SELF ATTR LINK`

See source error message number 89.

`***ERROR 254 LIMIT EXCEEDED: PROGRAM COMPLEXITY`

The program has too many complex expressions, cases, or procedures. Break it into smaller modules.

`***ERROR 255 LIMIT EXCEEDED: SYMBOLS`

See Appendix B for the correct limit.

NOTE

If a terminal error is encountered, program text beyond the point of error is not compiled. A terminal error message will appear at the point of error in the program listing.

14.2 Fatal Command Tail and Control Error Messages

Fatal command tail errors are caused by an improperly specified compiler invocation command or an improper control. The errors that can occur are as follows:

```
COMMAND TAIL TOO LONG
COMMAND TAIL BUFFER LIMIT EXCEEDED AT OR NEAR: xxx
ILLEGAL COMMAND TAIL SYNTAX OR VALUE
UNABLE TO PARSE COMMAND TAIL AT OR NEAR: xxx
ILLEGAL COMMAND TAIL SYNTAX OR VALUE
UNRECOGNIZED CONTROL IN COMMAND TAIL
INVOCATION COMMAND DOES NOT END WITH <hl>
ILLEGAL COMMAND TAIL SYNTAX
```

14.3 Fatal Input/Output Error Messages

Fatal input/output errors occur when the user incorrectly specifies a *pathname* for compiler input or output. These error messages are of the form:

```
PL/M-xxx ERROR -
FILE:
NAME:
ERROR:
COMPILATION TERMINATED
```

These errors also occur when the device runs out of space (e.g., the list file is larger than the available memory).

14.4 Fatal Insufficient Memory Error Messages

The fatal insufficient memory errors are caused by a system configuration with insufficient RAM memory to support the compiler.

The errors that can occur due to insufficient memory are as follows:

```
NOT ENOUGH MEMORY FOR COMPILATION
DYNAMIC STORAGE OVERFLOW
NOT ENOUGH MEMORY FOR CODE GENERATION
```

For PL/M-86 and PL/M-286, the following message can also occur:

```
NOT ENOUGH MEMORY FOR FINAL ASSEMBLY
```

14.5 Fatal Compiler Failure Error Messages

The fatal compiler failure errors are internal errors that should never occur. If you encounter such an error, please contact your Intel representative. The errors falling into this class are as follows:

```
***ERROR 89 COMPILER ERROR: BAD ERROR RECOVERY
***ERROR 90 COMPILER ERROR: MULTIPLE PARSE ARGS
***ERROR 92 COMPILER ERROR: PARSE ARG STACK UNDERFLOW
***ERROR 94 COMPILER ERROR: PARSE STACK UNDERFLOW
***ERROR 95 COMPILER ERROR: PARSE BUFFER OVERFLOW
***ERROR 97 COMPILER ERROR: SCOPE STACK UNDERFLOW
***ERROR 99 COMPILER ERROR: SEMANTIC UNDERFLOW
***ERROR 163 COMPILER ERROR: SEMANTIC UNDERFLOW
***ERROR 164 COMPILER ERROR: INVALID NODE
***ERROR 166 COMPILER ERROR: INVALID TREE
***ERROR 167 COMPILER ERROR: SCOPE STACK UNDERFLOW
***ERROR 175 COMPILER ERROR: INVALID OPERATION
***ERROR 177 COMPILER ERROR: REAL STACK UNDERFLOW
***ERROR 203 COMPILER ERROR: BAD LABEL ADDRESS
***ERROR 205 COMPILER ERROR: BAD CODE GENERATED
***ERROR 210 COMPILER ERROR: OBJECT MODULE
   GENERATION
```

For PL/M-86, error 210 is not used.

```
***ERROR 211 COMPILER ERROR: DEBUG SEGMENT SIZE
   OVERFLOW
***ERROR 212 COMPILER ERROR: ILLEGAL FIXUP
```

For PL/M-86, error 212 is not used.

```
***ERROR 230 COMPILER ERROR: INVALID INTERNAL TYPE
```

For PL/M-86, error 230 is not used.

```
***ERROR 251 COMPILER ERROR: INVALID OBJECT
***ERROR 252 COMPILER ERROR: SELF NAME LINK
***ERROR 253 COMPILER ERROR: SELF ATTR LINK
```

It is also possible to receive an UNKNOWN FATAL ERROR message.

14.6 Insufficient Memory Warning Messages

The following warnings may occur if there are too many symbols for symbol processing:

```
NOT ENOUGH MEMORY FOR FULL DICTIONARY LISTING
NOT ENOUGH MEMORY FOR ANY XREF PROCESSING
NOT ENOUGH MEMORY FOR FULL XREF PROCESSING
```

Appendixes

Tabs for
452161-001

APPENDIXES CONTENTS

A PL/M RESERVED WORDS AND PREDECLARED IDENTIFIERS

These are reserved words in PL/M-86, PL/M-286 and PL/M-386. They cannot be used as identifiers.

ADDRESS	INTEGER
AND	INTERRUPT
AT	LABEL
BASED	LITERALLY
BY	MINUS
BYTE	MOD
CALL	NOT
CASE	OR
DATA	PLUS
DECLARE	POINTER
DISABLE	PROCEDURE
DO	PUBLIC
DWORD	REAL
ELSE	REENTRANT
ENABLE	RETURN
END	SELECTOR
EOF	STRUCTURE
EXTERNAL	THEN
GO	TO
GOTO	WHILE
HALT	WORD
IF	XOR
INITIAL	

Additionally, the following are reserved words in PL/M-386.

CHARINT	OFFSET
HWORD	QWORD
LONGINT	SHORTINT

The following are identifiers for PL/M-86/286/386 built-in procedures and predeclared variables. If one of these identifiers is declared in a DECLARE statement, the corresponding built-in procedure or predeclared variable becomes unavailable within the scope of the declaration.

ABS
ADJUSTRPL
BLOCKINPUT
BLOCKINWORD
BLOCKOUTPUT
BLOCKOUTWORD
BUILDPTR
CARRY
CAUSEINTERRUPT
CMPB
CMPW
DEC
DOUBLE
FINDB
FINDRB
FINDRW
FINDW
FIX
FLAGS
FLOAT
GETREALERROR
HIGH
IABS
INITREALMATHUNIT
INPUT
INT
INWORD
LAST
LOCKSET
LENGTH
LOW
MOVB
MOVE
MOVRB
MOVRW
MOVW
NIL
OUTPUT
OUTWORD
RESTOREREALSTATUS
ROL
ROR
SAL
SAR
SAVEREALSTATUS
SCL
SCR
SELECTOROF
SETB
SETREALMODE
SETW
SHL
SHR
SIGN
SIGNED
SIZE
SKIPB
SKIPRB
SKIPRW
SKIPW
STACKBASE
STACKPTR
TIME
SIZE
UNSIGN
XLAT
ZERO

Additional reserved identifiers include:

PL/M-86

- INTERRUPT
- SETINTERRUPT

PL/M-286 and PL/M-386

CLEARTASKSWITCHEDFLAG	RESTOREINTERRUPTABLE
GETACCESSRIGHTS	SAVEGLOBALTABLE
GETSEGMENTLIMIT	SAVEINTERRUPTTABLE
LOCALTABLE	SEGMENTREADABLE
MACHINESTATUS	SEGMENTWRITABLE
OFFSETOF	TASKREGISTER
PARITY	WAITFORINTERRUPT
RESTOREGLOBALTABLE	

PL/M-386

CONTROLREGISTER	OUTHWORD
DEBUGREGISTER	SCANBIT
FINDHW	SCANRBIT
FINDRHW	SETHW
INHWORD	SHLD
MOVBIT	SHRD
MOVRBIT	SKIPHW
MOVHW	SKIPRHW
MOVRHW	TESTREGISTER

PL/M-386 with WORD16 control

BLOCKINDWORD	MOVD
BLOCKOUTDWORD	MOVRD
CMPD	OUTDWORD
FINDD	SETD
FINDRD	SKIPD
INDWORD	SKIPRD

B PL/M PROGRAM LIMITS

The user-program features that have limits are listed below with their maximum values:

Feature	Maximum PL/M-86	PL/M-286	PL/M-386
Size of LITERALLY string	unlimited*	unlimited*	unlimited*
Nesting of LITERALLY invocations	5	5	5
Nesting of INCLUDE controls	5	5	5
Number of nested procedures and DO cases	255	255	255
Number of labels on a statement	unlimited*	unlimited*	unlimited*
Nesting of blocks	18	18	18
Nesting of structures	32	32	32
Number of nested typed procedures	20	20	18
Number of elements in a factored list	64	64	64
Total number of members in a structure (at all levels)	128	128	128
Structure size	64K – 1	64K – 1	4G – 1
Numbers of characters in a line	128	128	128
Length of a string constant	255	255	255
Number of DO blocks in a procedure	65536	65536	65536
Number of cases in a DO CASE block	255	255	255
Number of active cases	255	255	255
Number of declared EXTERNAL items	unlimited*	unlimited*	**
Number of EXTERNAL items used	255	255	**
Number of procedures in a module	1015	1015	1015
Segment size	64K – 1	64K – 1	4G
Symbol capacity	See Note	See Note	2500

* Limited by the total size of the symbol table
** Limited by either the number of procedures or the number of symbols, or both.

PL/M-86/286

NOTE

The PL/M-86 and PL/M-286 compilers have a symbol capacity of approximately 5000 symbols. Of these, 800 are held in memory when the compiler has a partition size of 96K bytes. Any symbols over this amount will spill onto the workfiles disk, causing performance degradation. (If this happens, the compiler will display a message to the console.)

To increase performance, look at the dictionary summary (found in the compilation summary) to determine the amount of additional memory needed to avoid spilling to the disk. If another 64K bytes of memory is added to the compiler's partition (either by adding more memory to the system or by increasing its share of available memory), a total of 2300 symbols will then be held in memory.

For large programs or programs with many symbols, providing the compiler with more memory to work in will improve its performance.

end PL/M-86/286

C GRAMMAR OF THE PL/M LANGUAGE

This appendix lists the entire syntax of the PL/M language in Backus Naur Form (BNF). Since the semantic rules are not included here, this syntax permits certain constructions that are not actually allowed. The terminology used in the BNF syntax has been designed for convenience in constructing concise and rigorous definitions. Its appearance differs substantially from the main body of the manual.

The notations used here are slightly extended from standard BNF notations. An ellipsis (. . .) indicates that the syntactic element preceding it can be repeated indefinitely. The vertical bar (|) separates alternatives. Braces ({ }) indicate that only one of the items within them can be used. Brackets ([ ]) indicate options. When items are stacked vertically within brackets, only one of the items can be used.

C.1 Lexical Elements

C.1.1 Character Sets

```
<character> ::= <apostrophe> | <non-quote character>

<apostrophe> ::= '

<non-quote character> ::= <letter>
                        | <decimal digit>
                        | $
                        | <special character>
                        | blank

<letter> ::= <uppercase letter> | <lowercase letter>

<uppercase letter> ::= A | B | C | D | E | F | G | H | I |
                       J | K | L | M | N | O | P | Q | R |
                       S | T | U | V | W | X | Y | Z

<lowercase letter> ::= a | b | c | d | e | f | g | h | i |
                       j | k | l | m | n | o | p | q | r |
                       s | t | u | v | w | x | y | z

<decimal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special character> ::=  + | – | * | / | < | > | = | : | ; |
                         . | , | ( | ) | @ | _
```

C.1.2 Tokens

```
<token> ::= <delimiter>
          | <identifier>
          | <reserved word>
          | <numeric constant>
          | <string>
```

C.1.3 Delimiters

```
<delimiter> ::= <simple delimiter> | <compound delimiter>
```

```
<simple delimiter> ::=  + | − | * | / | < | > | = |  : | ; |
                        . |  , | ( | ) |@

<compound delimiter> ::= < > | < = | > = | : =
```

C.1.4 Identifiers

```
                                ┌                   ┐
                                │     <letter>      │
<identifier> ::= <letter>       │  <decimal digit>  │ . . .
                                │        $          │
                                └                   ┘

<reserved word> (For a list of reserved words, see Appendix A.)
```

C.1.5 Numeric Constants

```
<numeric constant> ::= <binary number>
                     | <octal number>
                     | <decimal number>
                     | <hexadecimal number>
                     | <floating point number>


<binary number> ::= <binary digit> ┌ <binary digit> ┐ . . .B
                                   └       $        ┘      Q

<octal number> ::= <octal digit> ┌ <octal digit> ┐ ┌ O ┐
                                 └       $       ┘ └ Q ┘

<decimal number> ::= <decimal digit> ┌ <decimal digit> ┐
                                     └        $        ┘ . . .[D]

<hexadecimal number> ::= <decimal digit> ┌ <hexadecimal digit> ┐ . . .H
                                         └          $          ┘

<floating point number> ::= <digit string> <fractional part>
                            [<exponent part>]

<fractional part> ::= [<.digit string>]
```

```
<exponent part> ::= E [+ | −] <digit string>

                                   ┌ <decimal digit> ┐
<digit string> ::= <decimal digit> │        $        │ . . .
                                   └                 ┘

<binary digit> ::= 0 | 1
<octal digit> ::= <binary digit> | 2 | 3 | 4 | 5 | 6 | 7
<decimal digit> ::= <octal digit> | 8 | 9
<hexadecimal digit> ::= <decimal digit> | A | B | C | D | E | F
```

C.1.6 Strings

```
<string> ::= ' <string body element> . . . '
<string body element> ::= <non-quote character> | ''
```

C.1.7 PL/M Text Structure: Tokens, Blanks, and Comments

```
                ┌   <token>     ┐
<pl/m text> ::= │ <separator>   │ . . .
                └               ┘
<separator> ::= blank | '' | <comment>

<comment> ::= /* [<character>] . . . */
```

C.2 Modules and the Main Program

```
<compilation> ::= <module> [EOF]
<module> ::= <module name> : <simple do block>
<module name> ::= <identifier>
```

C.3 Declarations

```
<declaration> ::= <declare statement> | <procedure definition>
```

C.3.1 DECLARE Statement

```
<declare statement> ::= DECLARE <declare element list>;
<declare element list> ::= <declare element>[,<declare element>]. . .
<declare element> ::= <factored element> | <unfactored element>

<unfactored element> ::= <variable element>
                       | <literal element>
                       | <label element>

<factored element> ::= <factored variable element>
                     | <factored label element>
```

C.3.2 Variable Elements

```
<variable element> ::= <variable name specifier>
                       [<array specifier>]
                       <variable type>
                       [<variable attributes>]
<variable name specifier> ::= <non-based name>
                            | <based name> BASED <base specifier>
<non-based name> ::= <variable name>
<based name> ::= <variable name>
<variable name> ::= <identifier>
<base specifier> ::= <identifier>[.<identifier>]

<variable attributes> ::= [PUBLIC] [<locator> <initialization>]
                        | [EXTERNAL] [<constant attribute>]
<locator> ::= AT(<expression>)

<constant attribute> ::= DATA

<array specifier> ::= <explicit dimension> | <implicit dimension>

<explicit dimension> ::= ( <numeric constant> )
<implicit dimension> ::= ( * )

<variable type> ::= <basic type> | <structure type>
```

PL/M-86/286

```
<basic type>::=INTEGER
             | REAL
             | POINTER
             | SELECTOR
             | BYTE
             | WORD
             | DWORD
             | ADDRESS
```

end PL/M-86/286

PL/M-386

```
<basic type>::=Address
             | BYTE
             | HWORD
             | DWORD
             | QWORD
             | CHARINT
             | OFFSET
             | SHORTINT
             | INTEGER
             | REAL
             | SELECTOR
             | POINTER
             | OFFSET
```

end PL/M-386

C.3.3 Label Element

```
                                            PUBLIC
<label element>::=<identifier> LABEL       EXTERNAL
```

C.3.4 Literal Elements

```
<literal element>::=<identifier> LITERALLY <string>
```

C.3.5 Factored Variable Element

```
<factored variable element>::=( <variable name specifier>
                                [,<variable name specifier>]. . .)
                                [<explicit dimension>] <variable type>
                                [<variable attributes>]
```

C.3.6 Factored Label Element

```
<factored label element>::=(<identifier>
                                                   ┌ PUBLIC   ┐
                      [,<identifier>]. . .) LABEL  └ EXTERNAL ┘
```

C.3.7 The Structure Type

```
<structure type>::= STRUCTURE (<member element>
                    [,<member element>]. . .)

<member element>::= <unfactored member>
                    | <factored member>

<unfactored member>::= member name [explicit dimension]
                       variable type

<member name>::= <identifier>

<factored member>::= (member name(,member name)...)
                     [explicit dimension] variable type
```

C.3.8 Procedure Definition

```
<procedure definition>::=<procedure statement>
                         [<declaration>. . .] [<unit>. . .] <ending>

<procedure statement>::=<procedure name> :PROCEDURE
                         [<formal parameter list>] [<procedure type>]
                         [<procedure attributes>];
<procedure name>::=<identifier>
<procedure type>::=<basic type>

<formal parameter list>::=(<formal parameter>[,<formal parameter>]. . .)

<formal parameter>::=<identifier>
```

```
                          ⎧⎡ <interrupt> ⎤⎫
                          ⎪⎢ EXTERNAL    ⎥⎪
<procedure attributes> ::= ⎨⎢             ⎥⎬
                          ⎪⎢ <interrupt> ⎥⎪
                          ⎪⎢ PUBLIC      ⎥⎪
                          ⎩⎣ REENTRANT   ⎦⎭
```

C.3.9 Attributes

C.3.9.1 AT

```
<locator> ::= AT (<expression>)
```

C.3.9.2 INTERRUPT

PL/M-86:

```
<interrupt> ::= INTERRUPT <numeric constant>
```

PL/M-286 and PL/M-386:

```
<interrupt> ::= INTERRUPT
```

C.3.9.3 Initialization

```
<initialization> ::= INITIAL (<initial value> [,<initial value>]. . .) | DATA
<initial value> ::= <expression> | <string>
```

C.4 Units

```
<unit> ::= <conditional clause>
         | <do block>
         | <basic statement>
         | <label definition> <unit>
<basic statement> ::= <assignment statement>
                    | <call statement>
                    | <goto statement>
                    | <null statement>
                    | <return statement>
                    | <microprocessor dependent statement>
```

```
<scoping statement> ::= <simple do statement>
                      | <do-case statement>
                      | <do-while statement>
                      | <iterative do statement>
                      | <end statement>
                      | <procedure statement>

<label definition> ::= <identifier>:
```

C.4.1 Basic Statements

C.4.1.1 Assignment Statement

```
<assignment statement> ::= <left part > = <expression>;
<left part> ::= <variable reference> [,<variable reference>] . . .
```

C.4.1.2 CALL Statement

```
<call statement> ::= CALL <simple variable> [<parameter list>];
<parameter list> ::= (<expression> [,<expression>] . . .)
<simple variable> ::= <identifier> | <identifier> . <identifier>
```

C.4.1.3 GOTO Statement

$$<\text{goto statement}> ::= \left\{\begin{matrix}\text{GOTO}\\ \text{GO TO}\end{matrix}\right\} <\text{identifier}>;$$

C.4.1.4 Null Statement

```
<null statement> ::= ;
```

C.4.1.5 RETURN Statement

```
<return statement> ::= <typed return> | <untyped return>

<typed return> ::= RETURN <expression>
<untyped return> ::= RETURN;
```

C.4.1.6 Microprocessor-Dependent Statements

```
<microprocessor dependent statement> ::= <disable statement>
                                       | <enable statement>
                                       | <halt statement>
                                       | <cause interrupt statement>

<disable statement> ::= DISABLE;
<enable statement> ::= ENABLE;
<halt statement> ::= HALT;
<cause interrupt statement> ::= CAUSE$INTERRUPT (numeric constant);
```

C.4.2 Scoping Statements

C.4.2.1 Simple DO Statement

```
<simple do statement> ::= DO ;
```

C.4.2.2 DO-CASE Statement

```
<do-case statement> ::= DO CASE <expression> ;
```

C.4.2.3 DO-WHILE Statement

```
<do-while statement> ::= DO WHILE <expression> ;
```

C.4.2.4 Iterative DO Statement

```
<iterative do statement> ::= DO <index part> <to part> [<by part>] ;
<index part> ::= <index variable> = <start expression>
<to part> ::= TO <bound expression>
<by part> ::= BY <step expression>
<index variable> ::= <simple variable>
<start expression> ::= <expression>
<bound expression> ::= <expression>
<step expression> ::= <expression>
```

C.4.2.5 END Statement

```
<end statement> ::= END [<identifier>];
```

C.4.2.6 Procedure Statement

```
<procedure statement> ::= <procedure name> : PROCEDURE
        [<formal parameter list>] [<procedure type>]
        [<procedure attributes>];
```

C.4.3 Conditional Clause

```
<conditional clause> ::= <if condition> <true unit>
        | <if condition> <true element> ELSE <false element>

<if condition> ::= IF <expression> THEN
<true element> ::= [<label definition> . . .] <do block>
                 | [<label definition> . . .] <basic statement>

<false element> ::= <unit>
<true unit> ::= <unit>
```

C.4.4 DO Blocks

```
<do block> ::= <simple do block>
             | <do-case block>
             | <do-while block>
             | <iterative do block>
```

C.4.4.1 Simple DO Blocks

```
<simple do block> ::= <simple do statement> [<declaration> . . .]
                      [<unit> . . .] <ending>

<ending> ::= [<label definition> . . .] <end statement>
```

C.4.4.2 DO-CASE Blocks

```
<do-case block> ::= <do-case statement> [<unit> . . .] <ending>
```

C.4.4.3 DO-WHILE Blocks

```
<do-while block> ::= <do-while statement> [<unit> . . .] <ending>
```

C.4.4.4 Iterative DO Blocks

```
<iterative do block>::= <iterative do statement> [<unit>. . .] <ending>
```

C.5 Expressions

C.5.1 Primaries

```
<primary>::= <constant>
           | <variable reference>
           | <location reference>
           | <subexpression>
<subexpression>::= ( <expression> )
```

C.5.1.1 Constants

```
<constant>::= <numeric constant> | <string>
```

C.5.1.2 Variable References

```
<variable reference>::= <data reference> | <function reference>
<data reference>::=  <name>[<subscript>] [<member specifier>]
<subscript>::= (<expression>)
<member specifier>::= .<member name>[<subscript>]
<function reference>::=  <name>[<actual parameters>]
<actual parameters>::=(<expression>[,<expression>]. . .)
<member name>::= <identifier>
<name>::=  <identifier>
```

C.5.1.3 Location References

```
<location reference>::= @ <constant list>
                      |  @<variable reference>

<constant list>::= (<constant>[,<constant>]. . .)
```

C.5.2 Operators

```
<operator> ::= <logical operator>
             | <relational operator>
             | <arithmetic operator>

<logical operator> ::= AND | OR | NOT | XOR

<relational operator> ::= < | > | < = | > = | < > | =
<arithmetic operator> ::=  + | − | PLUS | MINUS | * | / | MOD
```

C.5.3 Structure of Expressions

```
<expression> ::= <logical expression>
               | <embedded assignment>

<embedded assignment> ::= <variable reference> := <logical expression>

<logical expression> ::= <logical factor>
                       | <logical expression> <or operator> <logical factor>

<or operator> ::= OR | XOR

<logical factor> ::= <logical secondary>
                   | <logical factor> <and operator> <logical secondary>

<and operator> ::= AND
<logical secondary> ::= [<not operator>] <logical primary>
<not operator> ::= NOT
<logical primary> ::= <arithmetic expression>
      [<relational operator> <arithmetic expression>]

<relational operator> ::= < | >  | < = | > = | < > | =

<arithmetic expression> ::= <term>
                          | <arithmetic expression> <adding operator> <term>

<adding operator> ::= + | − | PLUS | MINUS
<term> ::= <secondary>
         | <term> <multiplying operator> <secondary>

<multiplying operator> ::= * | / | MOD
<secondary> ::= <unary minus> <primary> <unary plus>

<unary minus> ::= −
<unary plus> ::= +
```

D DIFFERENCES AMONG PL/M-86, PL/M-286, AND PL/M-386

D.1 Differences between PL/M-86 and PL/M-80

PL/M-86 differs from PL/M-80 in the following respects:

- Support for floating-point arithmetic
- Support for signed arithmetic
- Addition of REAL, INTEGER, POINTER and SELECTOR data types
- Addition of the @ location reference operator
- Support for nested structures
- Expanded set of built-in procedures

In addition, the PL/M-80 reserved word ADDRESS is replaced by the PL/M-86 reserved word WORD. PL/M-80 has only the BYTE and ADDRESS data types. However, PL/M-86 has the following data types: BYTE, WORD, DWORD, INTEGER, REAL, POINTER, and SELECTOR.

The PL/M-86 rules for expression evaluation are more complete than those of PL/M-80. Other differences stem from the ones noted here. For example, an iterative DO block operates differently if its index variable is an INTEGER variable.

D.2 Compatibility of PL/M-80 Programs and the PL/M-86 Compiler

PL/M-80 programs that operate correctly on an 8080 microprocessor can be recompiled with the PL/M-86 compiler to produce code that will run on an 8086 microprocessor. First edit the PL/M-80 source code as follows:

- PL/M-80 source code identifiers that are PL/M-86 reserved words must be changed.
- It is not necessary to change ADDRESS to WORD; ADDRESS is a PL/M-86 reserved word with the same meaning as WORD.

Note that where PL/M-86 programs would normally have POINTER variables and location references formed with the @ operator, PL/M-80 programs have ADDRESS (WORD) variables and location references formed with the dot operator. PL/M-80 usage is less restricted than PL/M-86 usage, because arithmetic operations can be used on WORD values. In general, the PL/M-86 compiler supports PL/M-80 usage to provide upward compatibility. Some restrictions affect the types of expressions that can be used in the AT attribute, the INITIAL and DATA initializations, and location references. See also the discussions of size controls and the dot and @ operators in this manual.

D.3 Differences between PL/M-286 and PL/M-86

PL/M-286 differs from PL/M-86 in the following respects:

- POINTER and SELECTOR variables cannot be assigned absolute (i.e., constant) values. Only the equals operator (=) can be used with POINTER variables. For SELECTOR variables the logical (AND, OR, NOT, XOR) and relational (< , > , < = , > = , < > , =) operators can be used.
- Access to the hardware flag register is provided with the built-in variable FLAGS.
- Four built-in functions have been added to support multiple byte and word input: BLOCKINPUT, BLOCKINWORD, BLOCKOUTPUT, and BLOCKOUTWORD (available to PL/M-86 via the MOD86/MOD186 control).
- The type of the STACKBASE variable has been changed from WORD to SELECTOR.
- New built-in procedures and functions have been added to support the 80286 hardware protection model.
- Interrupt procedures are no longer assigned numbers in the source program. (This is done by the 80286 system builder.) Interrupt procedures also cannot be called directly, and the SET$INTERRUPT and INTERRUPT$PTR built-ins have been removed.
- The memory array has been removed.

D.4 Compatibility of PL/M-86 Programs and the PL/M-286 Compiler

PL/M-86 programs that operate correctly on an 8086 microprocessor can be recompiled with the PL/M-286 compiler to produce code that will run on an 80286 microprocessor. The PL/M-86 source code must be edited as follows:

- Assignments to the STACKBASE built-in variable must be changed from WORD to SELECTOR.
- All absolute pointer and selector assignments must be changed. (Pointers can be assigned a zero value using the new built-in function NIL.) Also, relational operations on pointer and selector values for any operation other than equality and inequality must be changed.
- The interrupt numbers on all interrupt procedures must be deleted. Interrupt vectors will be assigned to these procedures by the 80286 system builder. Direct calls to interrupt procedures must also be changed.
- References to the SET$INTERRUPT, INTERRUPT$PTR, and MEMORY built-ins must be removed.

D.5 Differences between PL/M-386 and PL/M-286

PL/M-386 differs from PL/M-286 in the following respects:

- The string built-ins FIND, CMP, and SKIP return a value of 0FFFFFFFFH for the not found and string equal results.
- Support for 64-bit unsigned scalars.
- Support for 8-bit and 32-bit signed scalars.
- Addition of HWORD, CHARINT, and SHORTINT data types.
- ADDRESS is the same as OFFSET (and not as WORD as in PL/M-286).
- Support for WORD32 and WORD16 mapping for data type identifiers.
- Addition of the WORD32/WORD16 primary compiler controls, which ensure PL/M data type and language compatibility.
- MEDIUM and LARGE segmentation controls no longer indicate unique meaning to the compiler; MEDIUM is interpreted as SMALL and LARGE is interpreted as COMPACT except when LARGE is used to indicate a subsystem whose name is unknown at compile time.

- Several new built-in procedures and functions have been added to support the new data types (for example, CMPHW, BLOCKINHWORD; see Chapters 9 and 10), and some bit-string operations (for example, SCANBIT, MOVBIT).
- The built-ins CONTROL$REGISTER, DEBUG$REGISTER, and TEST $REGISTER have been added to support the 80386 microprocessor.

D.6 Compatibility of PL/M-286 Programs and the PL/M-386 Compiler

PL/M-286 source code can be compiled with the PL/M-286 compiler to produce code that will run on an 80386 microprocessor in 80286 microprocessor mode and interface with PL/M-386 code through INTERFACE(/286). In addition, PL/M-286 programs that operate on an 80286 microprocessor can be recompiled with the PL/M-386 compiler to produce code that will run on an 80386 microprocessor in the native 80386 microprocessor mode.

E ASCII CHARACTERS, HEX VALUES, AND PL/M CHARACTER SET

ASCII Character	HEX	PL/M Character?	ASCII Character	HEX	PL/M Character?
NUL	00	no	!	21	no
SOH	01	no	"	22	no
STX	02	no	#	23	no
ETX	03	no	$	24	yes
EOT	04	no	%	25	no
ENQ	05	no	&	26	no
ACK	06	no	'	27	yes
BEL	07	no	(	28	yes
BS	08	no	)	29	yes
HT	09	yes	*	2A	yes
LF	0A	yes	+	2B	yes
VT	0B	no	,	2C	yes
FF	0C	no	–	2D	yes
CR	0D	yes	.	2E	yes
SO	0E	no	/	2F	yes
SI	0F	no	0	30	yes
DLE	10	no	1	31	yes
DC1	11	no	2	32	yes
DC2	12	no	3	33	yes
DC3	13	no	4	34	yes
DC4	14	no	5	35	yes
NAK	15	no	6	36	yes
SYN	16	no	7	37	yes
ETB	17	no	8	38	yes
CAN	18	no	9	39	yes
EM	19	no	:	3A	yes
SUB	1A	no	;	3B	yes
ESC	1B	no	<	3C	yes
FS	1C	no	=	3D	yes
GS	1D	no	>	3E	yes
RS	1E	no	?	3F	no
US	1F	no	@	40	yes
space	20	yes	A	41	yes

ASCII Character	HEX	PL/M Character?	ASCII Character	HEX	PL/M Character?
B	42	yes	a	61	yes
C	43	yes	b	62	yes
D	44	yes	c	63	yes
E	45	yes	d	64	yes
F	46	yes	e	65	yes
G	47	yes	f	66	yes
H	48	yes	g	67	yes
I	49	yes	h	68	yes
J	4A	yes	i	69	yes
K	4B	yes	j	6A	yes
L	4C	yes	k	6B	yes
M	4D	yes	l	6C	yes
N	4E	yes	m	6D	yes
O	4F	yes	n	6E	yes
P	50	yes	o	6F	yes
Q	51	yes	p	70	yes
R	52	yes	q	71	yes
S	53	yes	r	72	yes
T	54	yes	s	73	yes
U	55	yes	t	74	yes
V	56	yes	u	75	yes
W	57	yes	v	76	yes
X	58	yes	w	77	yes
Y	59	yes	x	78	yes
Z	5A	yes	y	79	yes
[	5B	no	z	7A	yes
\	5C	no	{	7B	no
]	5D	no	|	7C	no
^(↑)	5E	no	}	7D	no
_	5F	yes	DEL	7F	no
`	60	no			

F LINKING TO MODULES WRITTEN IN OTHER LANGUAGES

F.1 Introduction

This appendix describes the calling conventions used by the 80[*x*]86 family of languages. These calling conventions are standardized so that a module written in PL/M can freely call procedures, subroutines, and subprograms in other modules written in other 80[*x*]86 languages.

The information in this appendix is not necessary to call PL/M procedures and functions from PL/M. See Chapter 8 for information about parameters and arguments.

The calling conventions and stack and register usage described in this appendix are needed to call ASM subroutines. Also, the corresponding data types listed at the end of this appendix are needed to write a subroutine that can pick up the data in the PL/M program. Refer to the ASM macro assembler operating instructions for more information about combining PL/M programs with ASM programs and for examples.

The easiest way to ensure compatibility between assembly-language subroutines that are combined with PL/M programs or procedures is to write a dummy procedure in PL/M. This procedure would have the same argument list and the same attributes as the desired assembly language subroutine. The PL/M procedure would then be compiled with the correct segmentation control, and the CODE control specified. This will produce a pseudo-assembly listing of the generated microprocessor code, which can then be copied to the prologue and epilogue of the assembly language subroutine.

With PL/M, separate modules can be written and compiled, and combined at a later time. This allows you to create separately tested modules that are combined after they are internally bug-free. Not all modules have to be in PL/M: you can choose the appropriate language for each module. Be sure to combine the modules properly with a binder or a linker in order to satisfy references to externals. Because the 80[*x*]86 languages (excluding C) follow the same calling sequence, control will pass to a called module correctly. (The C calling sequence is described in Section F.7. The standard calling sequence is described in the following section.) However, the called module might not be able to deal intelligently with the data passed to it since languages treat some data structures differently.

By specifying arguments in a reference to an external procedure, data is passed to the external procedure. The number of arguments and the order in which they are specified must match the number and order of the corresponding parameters in the external procedures declaration (see Chapter 8).

All arguments for parameters are passed on the microprocessor's stack, or the numeric coprocessor's register stack, in the order in which they were specified. For the 80386 microprocessor, the space occupied by a parameter pushed on the microprocessor's stack is always a multiple of four bytes. For the 8086 and 80286 microprocessors, the space occupied by a parameter pushed on the microprocessor's stack is always a multiple of two bytes. Functions return non-real values in a register, and REAL values on the top of the numeric coprocessor's register stack.

F.2 Calling Sequence

The calling sequence for each procedure activation places the procedure's actual parameters (if any) on the stack and then activates the procedure with a CALL instruction.

Parameters are placed on the microprocessor's stack or the numeric coprocessor's register stack in left-to-right order. Because the stack grows from higher locations to lower locations, the first parameter occupies the highest position on the stack, and the last parameter occupies the lowest position. Stack representation for the different PL/M parameters is described in Table F-1.

Table F-1 Stack Representation for PL/M Parameters

Parameter	8086 and 80286 Stack Representation	80386 Stack Representation
BYTE	Two bytes, with the higher byte undefined.	Four bytes, with the higher three bytes undefined.
CHARINT	N/A	Four bytes, with the higher three bytes undefined.
HWORD	N/A	Four bytes, with the high two bytes undefined.
SELECTOR	N/A	Four bytes, with the high two bytes undefined.

Table F-1 Stack Representation for PL/M Parameters (continued)

Parameter	8086 and 80286 Stack Representation	80386 Stack Representation
SHORTINT	N/A	Four bytes, with the high two bytes undefined.
WORD	Two bytes, with no undefined bytes.	Four bytes, with no undefined bytes.
OFFSET	N/A	Four bytes, with no undefined bytes.
INTEGER	N/A	Four bytes, with no undefined bytes.
REAL	N/A	Four bytes, with no undefined bytes.
DWORD	Four bytes, with the high 16 bits pushed first and the low 16 bits pushed second.	Eight bytes with the high 32 bits pushed first and the low 32 bits pushed second.

For the 8086 and the 80286 microprocessors, a POINTER parameter in the SMALL(ROM), COMPACT, MEDIUM, and LARGE cases consists of a selector and an offset. The 16-bit selector is pushed first, followed by the 16-bit offset.

For the 80386 microprocessor, a POINTER parameter in the SMALL(ROM) and COMPACT cases consists of a selector and an offset. The 16-bit selector is pushed first, followed by the 32-bit offset.

The left-most seven REAL parameters are passed on the numeric coprocessor's stack. If more than seven REAL parameters are present, the rest (after the left-most seven) are passed on the microprocessor's stack and are intermixed with the other non-real parameters in the order in which all parameters were declared.

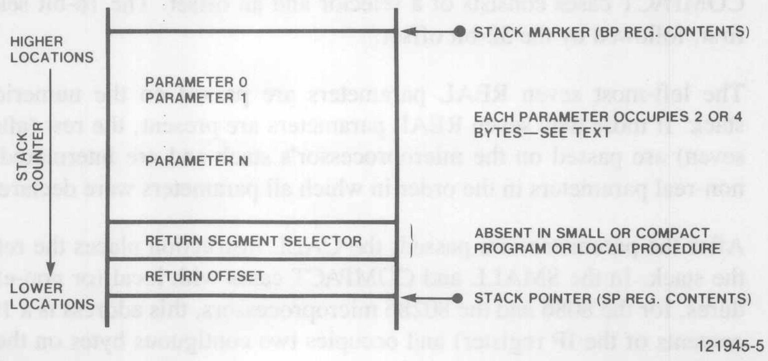
After the parameters are passed, the CALL instruction places the return address on the stack. In the SMALL and COMPACT cases with local (or non-exported) procedures, for the 8086 and the 80286 microprocessors, this address is a 16-bit offset (the contents of the IP register) and occupies two contiguous bytes on the stack. For the 80386 microprocessor, this address is a 32-bit offset (the contents of the EIP register) and occupies four bytes on the stack.

Specific to the 8086 and the 80286 microprocessors, in the MEDIUM and LARGE cases, and for procedures exported from COMPACT, the type of the return address depends on whether the procedure is local or public. The return address for a local procedure, like any return address for the SMALL case, is a 16-bit offset and occupies two contiguous bytes on the stack. For a public procedure in the MEDIUM or LARGE case, and for procedures exported from a subsystem, the return address is a POINTER value consisting of a selector and an offset; the return address is passed in the same way a POINTER parameter is passed. The 16-bit segment selector (contents of the CS register) is pushed first, then the 16-bit offset (IP register contents) is pushed.

Specific to the 80386 microprocessor, for procedures exported from a subsystem, the return address is a POINTER value consisting of a selector and offset; the return address is placed on the stack in the same way a POINTER parameter is passed. The 16-bit segment selector (contents of the CS register) is pushed first, then the 32-bit offset (EIP register contents) is pushed.

For all of the microprocessors, control is passed to the code of the procedure by updating the IP register for the 8086 and the 80286 microprocessors and the EIP register for the 80386 microprocessor. For procedures exported from a subsystem, the CS register is also updated.

Figure F-1 shows the stack layout at the point where the procedure gains control.



HIGHER LOCATIONS
STACK COUNTER
LOWER LOCATIONS
PARAMETER 0
PARAMETER 1
PARAMETER N
RETURN SEGMENT SELECTOR
RETURN OFFSET
STACK MARKER (BP REG. CONTENTS)
EACH PARAMETER OCCUPIES 2 OR 4 BYTES—SEE TEXT
ABSENT IN SMALL OR COMPACT PROGRAM OR LOCAL PROCEDURE
STACK POINTER (SP REG. CONTENTS)
121945-5

Figure F-1 Stack Layout at Point Where a Non-interrupt Procedure is Activated

F.3 Procedure Prologue

In compiling the procedure itself, the compiler inserts a sequence of instructions called the procedure prologue. The procedure prologue varies depending on the type of procedure being compiled as follows:

- If the procedure has the PUBLIC attribute and the program size is LARGE, or if it is exported from a subsystem, the content of the DS register is placed on the stack and is then updated to the data segment of the procedure. Additionally, for the 80386 microprocessor, ES is set to DS. (For the 8086 and the 80286 microprocessors, the DS register contains the segment selector for the current data segment; thus, this step implements the pairing of code and data segments in the LARGE case. It is not needed in the SMALL, COMPACT, and MEDIUM cases because the data segment does not change.)

- If any parameter of the procedure is referenced by a nested procedure, all parameters are copied from the stack to space reserved for them in the data segment.

- The stack marker offset (the 8086 and the 80286 microprocessors' BP register contents, and the 80386 microprocessor's EBP register contents) is placed on the stack, and the current stack pointer (the 8086 and the 80286 microprocessors' SP register contents, and the 80386 microprocessor's ESP register contents) is used to update the BP or EBP register.

- If the procedure has the REENTRANT attribute, space is reserved on the stack for any variables declared within the procedure (this does not include based variables, variables with the DATA attribute, or variables with the AT attribute).

Control then passes to the code compiled from the executable statements in the procedure body. Figure F-2 shows the stack layout at this point.

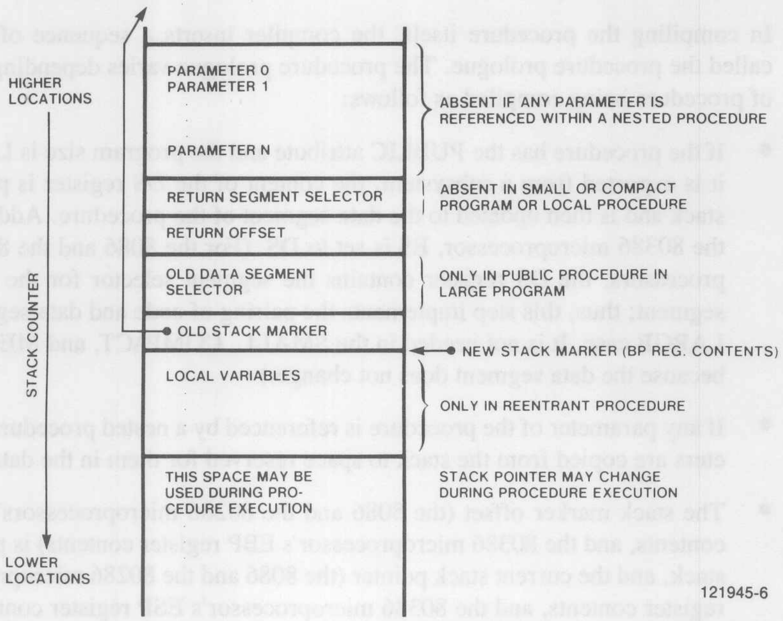

HIGHER LOCATIONS
STACK COUNTER
LOWER LOCATIONS
PARAMETER 0
PARAMETER 1
PARAMETER N
ABSENT IF ANY PARAMETER IS REFERENCED WITHIN A NESTED PROCEDURE
RETURN SEGMENT SELECTOR
ABSENT IN SMALL OR COMPACT PROGRAM OR LOCAL PROCEDURE
RETURN OFFSET
OLD DATA SEGMENT SELECTOR
ONLY IN PUBLIC PROCEDURE IN LARGE PROGRAM
OLD STACK MARKER
NEW STACK MARKER (BP REG. CONTENTS)
LOCAL VARIABLES
ONLY IN REENTRANT PROCEDURE
THIS SPACE MAY BE USED DURING PROCEDURE EXECUTION
STACK POINTER MAY CHANGE DURING PROCEDURE EXECUTION
121945-6

Figure F-2 Stack Layout During Execution of a Non-interrupt Procedure Body

F.4 Procedure Epilogue

To return from the procedure, the compiler inserts an instruction sequence called the epilogue. This accomplishes the following:

- If the compiler has used stack locations for temporary storage or local variables during procedure execution, the stack pointer (the SP register for the 8086 and the 80286 microprocessors, and the ESP register for the 80386 microprocessor) is loaded with the stack marker (the BP register for the 8086 and the 80286 microprocessors, and the EBP register for the 80386 microprocessor), discarding the temporary storage.
- The old stack marker is restored by popping the stored value from the stack into the BP or the EBP register.

- If the procedure has the PUBLIC attribute and the program size is LARGE or it is exported from a subsystem, the old data segment selector is restored by popping the stored value from the stack into the DS register. Additionally, for the 80386 microprocessor, ES is set to DS.
- For the 8086 and the 80286 microprocessor, if the program size is SMALL, the stored return address (a 16-bit offset) is popped into the IP register. Any parameters stored on the stack are discarded.
- For the 80386 microprocessor, the stored return address (a 32-bit offset) is popped into the EIP register. If the procedure is exported, the stored return address selector is also popped into the CS register. Any parameters stored on the stack are discarded.

Specific to the 8086 and the 80286 microprocessors, if the program size is MEDIUM or LARGE and the procedure is local, a return is performed using the actions previously described. If the program size is MEDIUM or LARGE and the procedure is public, the stored return-address offset from the stack is popped into the IP register and the return-address selector is popped into the CS register. Any parameters not stored on the stack are discarded.

F.5 Register Usage

Table F-2 provides a summary of the 8086 and the 80286 microprocessor register usage. Table F-3 provides a summary of the 80386 microprocessor register usage.

Table F-2 Summary of the 8086 and the 80286 Microprocessor Register Usage

Register	Must Preserve	Usage
AX	No	Return BYTE (AL), WORD, DWORD, INTEGER, and SELECTOR values; POINTER offset when using C language interface.
BX	No	Return POINTER offset values.
CX	No	—
DX	No	Return DWORD values; POINTER segment selector when using C language interface.
SP	Yes*	Stack pointer
BP	Yes	Stack marker

Table F-2 Summary of the 8086 and the 80286 Microprocessor Register Usage (continued)

Register	Must Preserve	Usage
SI	No (Yes when using C language interface.)	—
DI	No (Yes when using C language interface.)	—
FLAGS	No	—
CS	Yes	Called procedure's code segment.
DS	Yes	Caller's data segment.
SS	Yes	Caller's stack segment.
ES	No (Yes when using C language interface.)	Return POINTER segment selector.

*SP must be adjusted so that all arguments are removed from the stack on return.

Table F-3 Summary of the 80386 Microprocessor Register Usage

Register	Must Preserve	Usage
EAX	No	Return BYTE (AL), HWORD (AX), WORD, DWORD, CHARINT (AL), SHORTINT (AX), INTEGER, SELECTOR (AX), POINTER offset portion, and OFFSET.
EBX	No (Yes when using C language interface.)	
ECX	No	—
EDX	No	Return upper half of DWORD values, POINTER segment selector.
ESP	Yes*	Stack pointer
EBP	Yes	Stack marker
ESI	No (Yes when using C language interface.)	—
EDI	No (Yes when using C language interface.)	—
FLAGS	No	—

Table F-3 Summary of the 80386 Microprocessor Register Usage (continued)

Register	Must Preserve	Usage
CS	Yes	Called procedure's code segment.
DS	Yes	Caller's data segment.
SS	Yes	Caller's stack segment.
ES	Yes	Caller's data segment.
FS, GS	No	

*ESP must be adjusted so that all arguments are removed from the stack on return (except when using C language interface).

The numeric coprocessor's stack contains the first seven REAL arguments passed by the calling program. The numeric coprocessor's status word is unknown and does not need to be saved. If the status word is changed, the numeric coprocessor's mode word must be saved on entry and restored before exit.

If an assembly language subroutine alters the DS or SS registers, and expects to be called by a PL/M program, the subroutine must save the contents of these registers upon entry and restore them before returning to the PL/M program. Additionally, for the 80386 microprocessor, the CS and ES registers must be preserved by the called procedure.

PL/M uses the 8086 and the 80286 microprocessors' BP register or the 80386 microprocessor's ESP and EBP registers to address the stack. If a called assembly language subroutine uses the stack register, the subroutine must save the contents of the register on entry and restore the register's contents before returning control to the PL/M program. Before returning, the called subroutine must also adjust the 8086 and the 80286 microprocessors' SP register or the 80386 microprocessor's ESP register to remove all parameters from the microprocessor's stack. Additionally, specific to the 80386 microprocessor, the CS and ES registers must be preserved by the called procedure.

For the 8086 and the 80286 microprocessors, the AX, BX, CX, DX, SI, DI, and ES registers do not need to be preserved. For the 80386 microprocessor, the EAX, EBX, ECX, EDX, ESI, EDI, FS, and GS registers do not need to be preserved. A called subroutine can freely use these registers.

An assembly language program calling a PL/M procedure cannot expect the contents of the general-purpose registers (except BP and SP for the 8086 and the 80286 microprocessors, and EBP and ESP for the 80386 microprocessor) to be preserved. If the contents of these registers are needed, they must be saved prior to calling the PL/M procedure.

Table F-4 summarizes the 8086, 80286, and 80386 microprocessor registers used to hold simple data types that are returned by typed procedures.

NOTE

Returning a pointer to a gate, an unreadable code segment, or a segment that is not accessible at the caller's privilege level will cause a protection fault for the 80286 microprocessor. Also, an attempt to return a pointer to a segment that is not accessible at the called procedure's privilege level but is accessible at the caller's level will result in the caller receiving a null selector.

Table F-4 Registers Used to Hold Simple Data Types

8086 and 80286 Microprocessors Procedure Type	80386 Microprocessor Procedure Type	Register
BYTE	BYTE CHARINT	AL
WORD	HWORD SHORTINT	AX
DWORD		DX:AX
INTEGER		AX
	WORD OFFSET INTEGER	EAX
	DWORD	EDX:EAX
POINTER(SMALL)*		BX (AX when using the C language interface.)
POINTER(COMPACT)		ES:BX (DS:AX when using the C language interface.)
POINTER(MEDIUM)		ES:BX (DS:AX when using the C language interface.)

Table F-4 Registers Used to Hold Simple Data Types (continued)

8086 and 80286 Microprocessors Procedure Type	80386 Microprocessor Procedure Type	Register
POINTER(LARGE)		ES:BX (DS:AX when using the C language interface.)
	POINTER (SHORT, SMALL RAM)	EAX
	POINTER (LONG, COMPACT, SMALL ROM)	EDX:EAX
SELECTOR	SELECTOR	AX
REAL	REAL	Top of the numeric coprocessor's stack.

*Under the ROM option, the result is returned in ES:BX (DX:AX when using the C language interface).

F.6 Segment Name Conventions

Tables F-5 through F-7 summarize the segmentation of a subsystem under the program segmentation controls (SMALL and COMPACT for the 80386 microprocessor). The name of the segment in which each type of program section is stored (for each control, and for subsystems) is shown.

Table F-5 Summary of PL/M-86 Segment Names

Model	SubModel	Code	Data	Const	Stack
SMALL	IN DATA	CODE	DATA	CONST	STACK
	IN CODE	CODE	DATA	CONST	STACK
SMALL (subsystem)	IN DATA	sCODE	DATA	DATA	DATA
	IN CODE	sCODE	DATA	sCODE	DATA
COMPACT	IN DATA	CODE	DATA	CONST	STACK
	IN CODE	CODE	DATA	CONST	STACK
COMPACT (subsystem)	IN DATA	sCODE	sDATA	CONST	STACK
	IN CODE	sCODE	sDATA	CONST	STACK

Table F-5 Summary of PL/M-86 Segment Names (continued)

Model	SubModel	Code	Data	Const	Stack
MEDIUM	IN DATA	mCODE	DATA	CONST	STACK
	IN CODE	mCODE	DATA	CONST	STACK
LARGE	IN DATA	mCODE	mDATA	mDATA	STACK
	IN CODE	mCODE	mDATA	mCODE	STACK
LARGE (subsystem)	IN DATA	mCODE	mDATA	mDATA	STACK
	IN CODE	mCODE	mDATA	mCODE	STACK

Notes: sCODE denotes a segment name composed of the subsystem name and CODE.
sDATA denotes a segment name composed of the subsystem name and DATA.
mCODE denotes a segment name composed of the module name and CODE.
mDATA denotes a segment name composed of the module name and DATA.

Table F-6 Summary of PL/M-286 Segment Names

Model	SubModel	Code	Data	Const	Stack
SMALL	IN DATA	CODE	DATA	DATA	STACK
	IN CODE	CODE	DATA	CODE	STACK
SMALL (subsystem)	IN DATA	sCODE	DATA	DATA	DATA
	IN CODE	sCODE	DATA	sCODE	DATA
COMPACT	IN DATA	CODE	DATA	DATA	STACK
	IN CODE	CODE	DATA	CODE	STACK
COMPACT (subsystem)	IN DATA	sCODE	sDATA	sDATA	STACK
	IN CODE	sCODE	sDATA	sCODE	STACK
MEDIUM	IN DATA	mCODE	DATA	DATA	DATA
	IN CODE	mCODE	DATA	mCODE	DATA
LARGE	IN DATA	mCODE	mDATA	mDATA	STACK
	IN CODE	mCODE	mDATA	mCODE	STACK
LARGE (subsystem)	IN DATA	mCODE	mDATA	mDATA	STACK
	IN CODE	mCODE	mDATA	mCODE	STACK

Notes: sCODE denotes a segment name composed of the subsystem name and CODE.
sDATA denotes a segment name composed of the subsystem name and DATA.
mCODE denotes a segment name composed of the module name and CODE.
mDATA denotes a segment name composed of the module name and DATA.

Table F-7 Summary of PL/M-386 Segment Names

Model	SubModel	Code	Data	Const	Stack
SMALL	IN DATA	CODE32	DATA	DATA	DATA
	IN CODE	CODE32	DATA	CODE32	DATA
SMALL (subsystem)	IN DATA	S_CODE32	DATA	DATA	DATA
	IN CODE	S_CODE32	DATA	S_CODE32	DATA
COMPACT	IN DATA	S_CODE32	S_DATA	S_CODE32	STACK
	IN CODE	S_CODE32	S_DATA	S_CODE32	STACK
COMPACT (subsystem)	IN DATA	S_CODE32	DATA	S_DATA	STACK
	IN CODE	S_CODE32	DATA	S_CODE32	STACK

Notes: CODE32 denotes a segment name composed of CODE32.
DATA denotes a segment name composed of DATA.
S_CODE32 denotes a segment name composed of the subsystem name and CODE32.
S_DATA denotes a segment name composed of the subsystem name and DATA.

F.7 C Language Compatibility

The C-86/286 calling conventions, procedure prologue and epilogue, and register usage differ from other Intel 80286 languages. However the INTERFACE control, described in Section 11.2.7, allows C procedures to call procedures written in PL/M and vice versa.

The calling conventions, procedure prologue and epilogue, and register usage for C-386 language differ from the other Intel 80386 microprocessor languages. These difference are as follows:

- All parameters (real and non-real), are passed on the microprocessor's stack. The last parameter is pushed first and the first parameter is pushed last so that the first parameter is in the lowest memory location.
- An integral parameter that is less than four bytes must be zero or sign-extended, as required by the C language.
- The space occupied by a parameter pushed on the microprocessor stack is always a multiple of four bytes.

PL/M-386
(continued)

- Both short (floating-point) and long (double) real parameters are pushed as long real parameters, as required by the C language. Therefore, all real parameters passed from or to C procedures must be typed as 64-bit REAL in the PL/M-386 code.
- The calling procedure pops the parameters from the microprocessor stack after the called procedure has returned. Except when the called procedure is a function returning real results, the called procedure must not leave any entries in the numeric coprocessor stack.
- The ESP, EBP, CS, DS, ES, and SS registers should be preserved by the called procedure. (They are used for global storage.)
- The EBX, ESI, and EDI registers should also be preserved by the called procedure. These registers can be used by the caller for local data storage.
- The EAX, ECX, EDX, FS, and GS registers do not need to be preserved by the called procedure.

See the description of the INTERFACE control, in Chapter 11, for information on how to call C procedures from procedures written in PL/M-386, and vice versa.

end
PL/M-386

PL/M-86

F.8 Floating-point Libraries

This section deals with choosing the appropriate linkage specification to use the REAL math facility. The options are no use, PL/M-86 use only, or use of routines not written in PL/M-86. Choosing the correct linkage specification also depends on whether execution will use an actual numeric coprocessor or an 8087 emulator.

These linkage specifications make available the libraries of floating-point functions. The circumstances determining which library is appropriate are given in Table F-8.

Table F-8 Linkage Choices for REAL-Math Usage

Use of REAL Math Facility	Emulator or Numeric Coprocessor	Link-list Specifications
None	Neither	(none)
All floating point in PL/M-86 only	Emulator	E8087.LIB
With some modules that use floating point not in PL/M-86.	Emulator	E8087.LIB, E8087
Any	Numeric coprocessor	8087.LIB

The interface libraries do the following:

- 8087.LIB resolves external references inserted by the translator of an 8086 microprocessor program so that floating-point instructions will correctly invoke the numeric coprocessor. 8087.LIB is the library of floating-point functions written for the numeric coprocessor rather than for the 8087 emulator.

- E8087.LIB resolves such references to invoke the 8087 emulator software instead of the actual numeric coprocessor.

Emulation is performed by E8087.

- E8087 is the emulation routines library, which provides all the functions and features of a numeric coprocessor, except speed. Emulation is invoked automatically using interrupts 20 through 31. The emulator occupies approximately 16K bytes of code space.

The 8087 emulator processes exceptions exactly as the numeric coprocessor does. However, if the microprocessor/numeric coprocessor implementation includes an external interrupt masking device such as an 8259A, the effect of this external device cannot be simulated by the 8087 emulator. An interrupt 16 occurs after execution of any instruction when the emulated interrupt is active and the microprocessor interrupt is enabled, even if the 8259A is disabled.

PL/M-86
(continued)

To locate the 8087 emulator at a specified memory location:

- Locate the read-only code by referring to class AQMCODE in the LOC86 invocation.
- Locate the read-write data area by referring to class AQMDATA.

The 8087 emulator uses the 8086 microprocessor's interrupts 20 through 31. If the program uses any REAL variables, absolute memory locations 50H through 7FH in the 8086 microprocessor's final located module will contain the necessary code for these interrupts in the interrupt vector. These memory locations should not be used for any other purpose.

end
PL/M-86

G RUN-TIME INTERRUPT PROCESSING

G.1 General Information

Interrupts can be initialized when the CPU receives a signal on its maskable interrupt pin from a peripheral device, or when control is transferred to an interrupt vector by the CAUSE$INTERRUPT statement. If the program runs under an operating system that traps interrupts, the information in this appendix may not be applicable.

Note that the CPU does not respond to the interrupt signal unless interrupts are enabled. The compiler does not generate any code to enable or disable interrupts at the start of the main program.

If interrupts are enabled and vectored through an interrupt gate, the following actions take place:

1. The CPU completes any instruction currently in execution.
2. The CPU issues an acknowledge interrupt signal and waits for the interrupting device to send an interrupt number.
3. The CPU flag register is placed on the stack (occupying two bytes of stack storage).
4. Interrupts are disabled by clearing the IF flag.
5. Single stepping is disabled by clearing the TF flag.
6. The CPU activates the interrupt procedure corresponding to the interrupt number sent by the interrupting device.
7. When that procedure terminates, the stack is automatically restored to the state it was in when the interrupt was received, and control returns to the point where it was interrupted.

The mechanism for this activation and restoration are described in the following sections. If interrupts are vectored through a trap gate, the fourth step is not performed; if they are vectored through a task gate, all seven steps are replaced by a task switch.

G.2 The Interrupt Vector

If the NOINTVECTOR control is not specified, an interrupt vector entry is automatically generated by the compiler for each interrupt procedure. Collectively, the interrupt vector entries form the interrupt vector. If NOINTVECTOR is specified, the interrupt vector must be supplied as explained in Chapter 11.

The interrupt vector is an absolutely located array of POINTER values beginning at location 0. Thus the nth entry is at location 4*n and contains the location of the procedure declared with the INTERRUPT *n* attribute.

Note that the first and second bytes of each entry contain an offset, while the second two bytes contain a segment address. The entries are always four-byte pointers, and the segment address is always used in transferring to the interrupt procedure, even if the program size is SMALL.

The CPU uses the interrupt vector entry to make a long indirect call to activate the appropriate procedure. At this point, the current code segment address (CS register contents) and instruction offset (IP register contents) are placed on stack.

At the point where the procedure is activated, the stack layout is as shown in Figure G-1.

end
PL/M-86

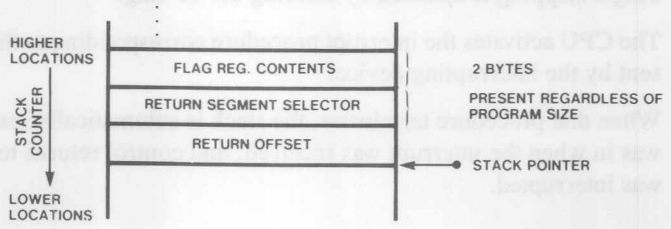


Figure G-1 Stack Layout at Point Where an Interrupt Procedure Gains Control

G.3 The Interrupt Descriptor Table

The interrupt descriptor table (IDT) contains descriptors that vector interrupts, traps, and protection exceptions to their respective handling routines.

These descriptors are called gates; they can be either interrupt gates, trap gates, or task gates. Interrupt gates and trap gates point to a particular entry point in the address space of the interrupted user (i.e., to an interrupt procedure). Task gates point to an interrupt processing task state segment (TSS).

The system builder is used to set up the IDT and to assign numbers to vector the individual gates to the appropriate interrupt procedure or task. For more information, see the system builder user's guide.

The IDT can hold up to 256 gates. Gates 0 through 31 are reserved for internal use.

G.3.1 Procedures and Tasks

For the 80286 microprocessor, when an interrupt is vectored through an interrupt gate, all registers must be pushed onto the stack (see section G.4), interrupts are automatically saved in the TSS, and the microprocessor's registers are loaded from the TSS of the interrupt task. Thus, no explicit register saving is necessary. Interrupts are enabled or disabled depending on the setting of the flags in the interrupt task's TSS during its execution (unless explicitly changed). This enables the interruption of the interrupt task. Note, however, that if the same interrupt that is currently being handled occurs, a protection violation will occur.

Similarly, for the 80386 microprocessor, when an interrupt is vectored through an interrupt gate, all registers must be pushed onto the stack (see section G.4), and interrupts are automatically disabled. (Interrupts must be explicitly enabled.) The interrupt procedure then begins execution. The interrupt procedure ends with an IRET instruction that acts as a normal return. Hence, execution starts at the beginning of a procedure each time it is entered.

Specific to the 80836 microprocessor, the interrupt process differs for an interrupt vectored through a task gate. The registers for the interrupted task are saved in the

PL/M-286/386
(continued)

TSS, and the microprocessor's registers are loaded from the TSS of the interrupt task. Thus, no explicit register saving is necessary. Interrupts are enabled or disabled depending on the flag settings in the interrupt task's TSS during execution of the interrupt task (unless explicitly changed). This enables interruption of the interrupt task. However, a protection violation will occur if an interrupt task is busy and an attempt is made to vector through the busy interrupt task.

For both the 80286 and the 80386 microprocessors, the interrupt task also ends with an IRET instruction, but in this case it acts as a task switch, saving the status of the outgoing interrupt task in memory. When the task is re-entered, execution continues at the first instruction after the IRET instruction.

end
PL/M-286/386

G.4 Interrupt Procedure Prologue and Epilogue

An interrupt procedure begins by declaring its name and its PUBLIC or EXTERNAL attribute. The following interrupt procedure declaration is the correct form for PL/M-286 and PL/M-386:

```
HANDLER: PROCEDURE INTERRUPT PUBLIC;
```

For PL/M-86:

```
HANDLER: PROCEDURE INTERRUPT 207 EXTERNAL;
```

This alerts the compiler to create a code prologue appropriate to a routine that will, in general, be invoked by interrupts.

PL/M-86/286

For the 8086 and the 80286 microprocessors, at the beginning of each interrupt procedure, the interrupt procedure prologue inserted by the compiler accomplishes the following tasks:

1. Pushes the CPU registers onto the stack in the following order: AX, CX, DX, BX, SP, BP, SI, DI.
2. Pushes the ES register contents onto the stack.
3. Pushes the DS register contents onto the stack.
4. Loads the DS register with a new data segment address taken from the current code segment (i.e., the segment containing the interrupt procedure).

PL/M-86/286
(continued)

5. Pushes the BP register contents onto the stack.

6. Loads the stack marker (BP register) with the address taken from the current stack pointer (SP register).

7. If the procedure has the REENTRANT attribute, space is reserved on the stack for any local variables declared in the procedure.

However, if the procedure is REENTRANT and contains local variables, steps five, six, and seven will be replaced by the ENTER instruction.

end PL/M-86/286

PL/M-386

For the 80386 microprocessor, at the beginning of each interrupt procedure, the interrupt procedure prologue inserted by the compiler accomplishes the following tasks:

1. Pushes the CPU registers onto the stack in the following order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

2. Pushes the ES, FS, GS, and DS register content on the stack.

3. If the interrupt procedure has the PUBLIC attribute, and if it is exported from a subsystem, the contents of the DS register is placed on the stack and is then updated to the data segment of the procedure. In addition, ES is set to DS.

4. The stack marker offset (EBP register contents) is placed on the stack, and the current stack pointer (ESP register contents) is used to update the EBP register.

5. If the procedure has the REENTRANT attribute, space is reserved on the stack for any variables declared within the procedure. (This does not include based variables, variables with the DATA attribute, or variables with the AT attribute.)

end PL/M-386

NOTE

The compiler may temporarily use the DS register (and, for the 80386 microprocessor, the ES register) in some cases (e.g., string built-ins), but always restores it. Take care to note this possibility when writing an interrupt procedure in assembly language.

At the point where the interrupt procedure prologue gains control, the stack layout is as shown in Figure G-1.

After the interrupt procedure prologue is executed (at the point where the code compiled from the procedure body gains control), the stack layout is as shown in Figure G-2.

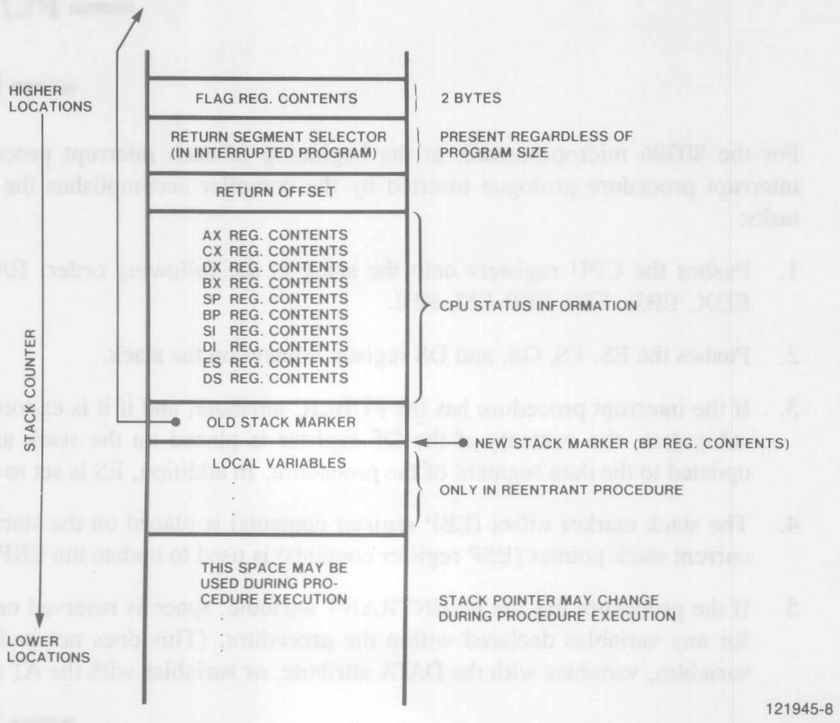


Figure G-2 Stack Layout during Execution of Interrupt Procedure Body

The return from the procedure body is called the interrupt procedure epilogue; it restores the stack to the state it was in before the interrupt occurred. The interrupt procedure epilogue contains the following steps:

1. If the compiler has used stack locations for temporary storage or local variables during procedure execution, the stack pointer (the SP register for the 8086 and the 80286 microprocessors and the ESP register for the 80386 microprocessor) is loaded with the current stack marker (the BP register for the 8086 and the 80286 microprocessors and the EBP register for the 80386 microprocessor) discarding the temporary storage.

2. The old stack marker is restored by popping the stored value from the stack into the EP or EBP register.
3. The old data segment is restored by popping the stored value from the stack into the DS register. For the 80386 microprocessor, this step will occur only if the procedure has a PUBLIC attribute and it is exported from a subsystem.

4. For the 8086 and the 80286 microprocessors, the stack is popped into the ES register.
5. For the 8086 and the 80286 microprocessors, the stack is popped into the CPU registers in the following order: DI, SI, BP, BX, DX, CX, AX. Note that the SP register value is discarded.

   For the 80386 microprocessor, the stack is popped into the CPU registers in the following order: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX. Note that the ESP register value is discarded.
6. An IRET instruction is executed to return from the interrupt procedure restoring the IP or EIP, CS, and the flag register contents from the stack.

For the 8086 and the 80286 microprocessors, if the procedure is REENTRANT and has local variables, steps one and two will be replaced by the LEAVE instruction.

At this point, the stack has been restored to the state it was in before the interrupt occurred, and processing continues normally.

PL/M-86

G.5 Writing Interrupt Vectors Separately

In some cases it may be desirable to write the interrupt vector separately (in PL/M or assembly language). Do so by using NOINTVECTOR to prevent generation of an interrupt vector by the compiler. Then link separately created interrupt vectors into the program.

Creation of a separate explicit vector requires some care. The @ operator in PL/M provides access to a procedure's normal (i.e., called) entry point, not to its interrupt entry point. The interrupt entry point first saves the status of the interrupted program before invoking the interrupt procedure through its normal entry point. The exact length of these operations depend on the compilation options chosen, the attributes of the interrupt procedure, and the version of the compiler being used. Use the built-in function INTERRUPT$PTR during execution to return the actual interrupt entry point. Discussion of this function appears in Chapter 9.

For example, suppose that two modules for a multimodule program are developed separately. Both use interrupt procedures, but when the modules are written the assignment of interrupt numbers to the various interrupt procedures has not been determined.

The two modules are therefore compiled with the NOINTVECTOR control. When this is done, the *n* in an INTERRUPT *n* attribute is ignored; since normally it would only be used to put the procedure's entry in the proper location within the interrupt vector.

Later, when the program is combined, a separately created interrupt vector can be linked in. Within this interrupt vector, the placement of the entry for a given interrupt procedure determines which interrupt number will activate that procedure.

Similarly, a library of interrupt procedures could all be compiled with NOINT-VECTOR. Any program could then have any of these procedures linked in, with a separately created interrupt vector.

Use the built-in procedure SET$INTERRUPT during execution to create the correct interrupt vector for each interrupt routine. This procedure is discussed in Chapter 9.

end
PL/M-86

G.6 Interrupt Tasks

A task on the microprocessor is a single thread of execution; that is, a stream of instructions and data with a task state image. The task state image is made up of the contents of the task registers, the task's status word, and the virtual locations of the task's instructions and data segments.

Tasks are initiated with a task switch operation. The CPU stores the task state image of the outgoing task (held in the processor registers) in memory, and loads the task state image of the incoming task into task registers. Because all the registers are reloaded and a new address space is entered, it is impossible to jump directly from one task to another.

Interrupt tasks are frequently written as one loop. At the beginning is the code needed to initialize the task, followed by the steps needed to handle the interrupt. Call the WAIT$FOR$INTERRUPT built-in procedure (see Chapter 10) to generate an IRET instruction. When the task is activated again, execution continues at the instruction following the IRET, with all the registers unchanged. At the end of the interrupt task, use a GOTO statement to loop back to the top of the interrupt task. Thus, an interrupt task never terminates, unless an operating system function removes the task.

Use of the WAIT$FOR$INTERRUPT procedure is demonstrated in the following example. This task is designed to handle messages that arrive in pieces, each one being preceded by an interrupt.

```
TASK: DO
      DECLARE local variables;
      local procedures;

NEW$MESSAGE:
      CALL INITIALIZE$MESSAGE$PROCESSING;
      DO FOREVER;
          CALL WAIT$FOR$INTERRUPT;
        /* IRET to wait for next interrupt, which continues here */
```

PL/M-286/386
(continued)

```
                CALL PROCESS$PIECE$OF$MESSAGE;
                IF LAST$PIECE$OF$MESSAGE$ THEN DO;
                   CALL TERMINATE$MESSAGE$PROCESSING;
                   CALL WAIT$FOR$INTERRUPT;
                              /* IRET to wait for start of next message */
                   GOTO NEW$MESSAGE
                END;
            END;
         END TASK;
```

end
PL/M-286/386

G.7 Exception Conditions in REAL Arithmetic

Six exception conditions can occur during normal numeric operations:

- Invalid operation
- Denormalized operand
- Zero divide
- Overflow
- Underflow
- Precision

These exceptions are discussed in the following sections. In each case, only a few of the possible causes are described because most are not likely to occur with PL/M usage. To perform sophisticated numeric processing of extreme precision and flexibility, refer to the microprocessor-specific programmer's reference manual.

The six exceptions and their default responses are summarized in Table G-1.

As the following sections indicate, the masked, default response to most exceptions will provide the least abrupt, most appropriate action for PL/M applications. Many real math exceptions that occur in other processors will not occur with the numeric coprocessor because of the extended range of intermediate results it holds. The soft recovery of gradual underflow (described in the denormal exception section) also extends the range of permissible execution rather than reporting a hard-failure condition.

Table G-1 Exception and Response Summary

Exception	Masked Response	Unmasked Response
Invalid Operation	If one operand is NAN, return it; if both are NANs, return NAN with larger absolute value; if neither is NAN, return indefinite NAN.	Request interrupt. (Numeric coprocessor stack unchanged.)
Zero divide	Return infinity signed with "exclusive or" of operand signs.	Request interrupt. (Numeric coprocessor stack unchanged.)
Denormalized	Memory operand: proceed as usual. Register operand: convert to valid unnormal, then reevaluate for exceptions.	Request interrupt. (Numeric coprocessor stack unchanged.)
Overflow	Return properly signed infinity.	Register destination: adjust exponent, store result, (see note) request interrupt. Memory destination: request interrupt.
Underflow	Denormalize result.	Register destination: adjust exponent, store result, (see note) request interrupt. Memory destination: request interrupt.
Precision	Return rounded result.	Return rounded result, request interrupt.

Note: On overflow, 24,576 decimal is subtracted from the true result's exponent. This forces the exponent back into range and enables a user exception handler to ascertain the true result from the adjusted result that is returned. On underflow, the same constant is added to the true result's exponent.

Programmers who use the recommended setting of the REAL mode word (see Chapter 10) need to handle only the invalid exception. Study of the other exception conditions is advised, however, to gain a general understanding of their use.

G.7.1 Invalid Operation Exception

This exception generally indicates a program error. It could be caused by referencing an uninitialized REAL variable or by referencing a location that does not contain a REAL value (as might occur with an out-of-range subscript for a REAL array).

Attempting to take the square root of a negative number or to store a number too large for integer format would also generate this exception.

Another interpretation of this exception is stack error. This may be caused by failing to restore the math unit status before returning from an interrupt routine that had saved the status. Another cause is the generation of more than eight intermediate results during REAL arithmetic, which can arise if REAL procedure function calls are nested too deeply. The compiler ensures that no single procedure does this, but cannot check what may occur only at run time. This exception can also occur when REAL functions (typed procedures) are used as operands within longer REAL expressions. For example:

```
    DELTA$1 = ALPHA * (BETA/GAMMA) + CHI (PSI, RHO, PI)
```

where all these names are typed REAL and CHI is some previously declared REAL function having three parameters.

The following is less likely to cause an exception condition:

```
    EPS = CHI (PSI, RHO, PI)
    DELTA$1 = ALPHA * (BETA/GAMMA) + EPS
```

because all REAL arithmetic is performed using the numeric coprocessor's stack, which has eight registers. The first seven REAL parameters supplied in procedure calls are placed on this stack. If the procedure is typed (i.e., invoked as a function), it can be imbedded as one operand within a longer REAL expression.

Because the evaluation of such an expression also involves the use of this stack for prior and subsequent arithmetic operations, stack overflow may occur. This overflow amounts to unpredictable destruction of original parameters or intermediate results. It becomes more likely as the complexity of REAL expressions containing REAL functions is increased. Thus, it is safer to use an assignment statement first to store the function's value in a real variable; then use that variable in the larger expression.

If stack error might apply, modify the code for the effected procedures to call the built-in procedures SAVE$REAL$STATUS and RESTORE$REAL$STATUS as their first and last operations, respectively.

The masked (default) response is to set the result to one of the special bit patterns called Not-A-Number (NANs), usually the indefinite value, the smallest NAN representable in the specified precision. It also sets bit 0 of the REAL error byte.

If bit 0 of the REAL mode word is 0 (invalid exception unmasked), an interrupt occurs, transferring control to the user-written interrupt handler.

G.7.2 Denormal Operand Exception

This condition arises when numeric operations have resulted in a number whose exponent is literally zero and whose significand is non-zero, or have resulted in a number whose significand does not begin with a one. Denormals usually arise in response to masked underflow. Gradual underflow is the masked, default response to underflow. A small denormal added to a large normal REAL number can give an acceptable, in-range answer if the denormal exception is masked. In practice, denormals are very rare since intermediate results are kept in temporary real format (15-bit exponent).

This condition causes bit 1 of the REAL error byte to be set to 1. If bit 1 of the REAL mode word is 1, the response described previously occurs. If bit 1 is 0, an interrupt occurs, transferring control to the user-written interrupt handler.

G.7.3 Zero Divide Exception

This condition arises when in the course of some REAL computation a divisor turns out to be zero. The masked response, when bit 2 of the REAL mode word is 1, is to return infinity appropriately signed. If bit 1 is 0, an interrupt occurs, giving control to the user-written interrupt handler. In either case, bit 2 of the REAL error byte is set to 1.

G.7.4 Overflow Exception

This error occurs when a real result is too large for the format in use. For assigning to REAL scalar types, it occurs if the result is greater than about 3.37 x 10**38. For intermediate REAL computations, it occurs if the result is greater than about 10**4932. The overflow exception can arise during assignment, addition, subtraction, multiplication, division, or conversion to integer.

The masked, default response (bit 3 of REAL mode word = 1) is to return infinity (signed if affine mode is set) and set bit 3 of the REAL error byte to 1. Unmasked overflow must go through a user-written interrupt handler.

G.7.5 Underflow Exception

Underflow exception is caused by an exponent too small for the format in use. For REAL assignments, it occurs if the exponent is less than −127; and for intermediate results if the exponent is less than −16383. Underflow can be caused by the same type of REAL operations as overflow.

The masked, default response (bit 4 of REAL mode word = 1) is to use the denormal number created by adjusting the very small result. It adjusts the significand, moving significant digits off to the right and raising the exponent until the latter becomes non-zero. For example, with single precision values, a 24-bit significand of .01 with an exponent of zero implies the number 1 x 2** – 129 because a zero exponent in this format means – 127. If the denormal exception is masked, this number would be adjusted into a significand of .001 with an exponent of 1 (i.e., 0.001 x 2** – 126), prior to operation. This number would then be available for use in subsequent REAL operations that might yield valid results. Zero is returned if it is the rounded result. Bit 4 of the REAL error byte is set to 1. Unmasked underflow must go through a user-written interrupt handler.

G.7.6 Precision Exception

This error occurs when the result of an operation is inexact (i.e., rounded) or when an overflow exception occurs. No special action is performed by a masked response (bit 5 of REAL mode word = 1) other than setting bit 5 of the REAL error byte. Unmasked response is as chosen by the user.

G.8 Writing a Procedure to Handle REAL Interrupts

This section partially summarizes the information pertaining to interrupts, floating-point usage, and procedures. (Additional facilities for handling REAL interrupts may be provided by the operating system, or can be performed with the system builder.)

An interrupt-handling procedure for PL/M-286 may, for example, begin as follows:

```
    HANDLER: PROCEDURE INTERRUPT PUBLIC;
```

If HANDLER will do any REAL arithmetic or assignments, its first executable statements should be of the form:

```
ERR$INFO = GET$REAL$ERROR; /* must declare ERR$INFO$ BYTE earlier */
```

or:

```
CALL SAVE$REAL$STATUS (@Local_Save_Area); /* also declare earlier */
```

Each procedure clears the error byte. The latter procedure also clears out the REAL stack. Thus, after either procedure is used, the REAL error byte no longer contains the flagged cause of the exception condition that invoked HANDLER.

Using SAVE$REAL$STATUS is a way of avoiding possible stack errors from cumulative usage. This enables errors in HANDLER to be detected independently of the originating exception condition. It also enables HANDLER to restore the state of the interrupted procedure despite HANDLER's own use of the REAL facility. SAVE$REAL$STATUS also makes available all the information regarding the state of the numeric coprocessor exceptions, stack, and operations, as shown in the following paragraph.

Thus, the beginning of a typical routine for the 80286 microprocessor to handle REAL exception conditions could look like this:

```
HANDLER: PROCEDURE INTERRUPT PUBLIC;

         DECLARE ERR$INFO BYTE;
         DECLARE LOCAL$SAVE$AREA (94) BYTE;
         ERR$INFO = GET$REAL$ERROR;
```

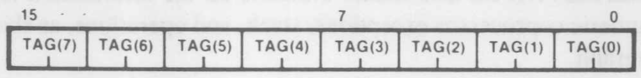
or, to perform extensive manipulations on the save area, declare a structure permitting access to the save area's component parts by name and/or byte, as follows:

```
HANDLER: PROCEDURE INTERRUPT PUBLIC;

         DECLARE ERR$INFO BYTE;
         DECLARE SAVE$AREA STRUCTURE (
                                  CONTROL(2)      BYTE,
                                  STATUS(2)       BYTE,
                                  TAG             WORD,
                                  INSTR_OFF       WORD,
                                  INSTR_SEL       SELECTOR,
                                  OPERAND_OFF     WORD,
                                  OPERAND_SEL     SELECTOR,
                                  STACK_TOP(5)    WORD,
                                  STACK_ONE(5)    WORD,
                                  STACK_TWO(5)    WORD,
                                  STACK_3 (5)     WORD,
                                  STACK_4 (5)     WORD,
                                  STACK_5 (5)     WORD,
                                  STACK_6 (5)     WORD,
                                  STACK_7 (5)     WORD);

         CALL SAVE$REAL$STATUS (@SAVE_AREA);

         ERR$INFO = SAVE_AREA.STATUS(0);
```

NOTE

To make use of the TAG word, use the masks and shifts to access the individual fields shown in Figure G-3.

Call either the SAVE$REAL$STATUS procedure or the GET$REAL$ERROR function, but not both. If the extra information gained by the SAVE is not needed (i.e., only the exceptions are needed), use the GET$REAL$ERROR function. If both are called, the second call will produce incorrect results.



Tag values:
00 = Valid (Normal or Unnormal)
01 = Zero (True)
10 = Special (Not-A-Number, ∞, or Denormal)
11 = Empty

121623-12

Figure G-3 Tag Word Format

The rest of HANDLER can perform any actions deemed appropriate. This is an application dependent decision. Among the possibilities:

- Incrementing an exception counter for later display
- Printing diagnostic data (e.g., the contents of SAVE$AREA)
- Aborting further execution of the calculation causing exception
- Aborting all further execution

The format of the LOCAL_SAVE_AREA as it is filled by the save procedure is shown in Figures G-4 and G-5.

The final action prior to returning (if desired) to the interrupted procedure is to restore the status of the REAL math unit:

```
CALL RESTORE$REAL$STATUS (@LOCAL_SAVE_AREA);
```

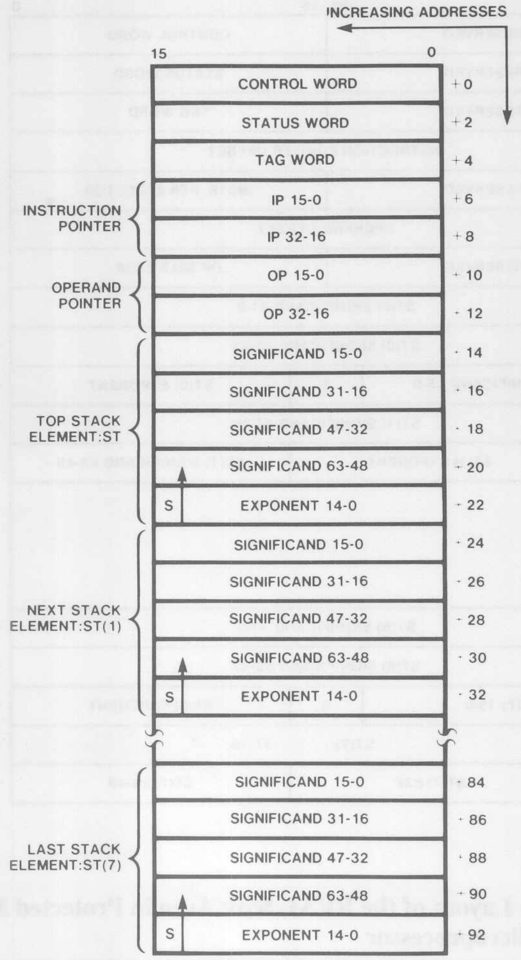
However, if GET$REAL$ERROR is not used prior to the SAVE$REAL$STATUS call, the local save area will contain the original contents of the error byte. Under these circumstances, first clear the lower byte of the saved status word before the RESTORE statement to avoid retriggering the same exception that invoked HANDLER in the beginning.

To do so, use a command of the form:

```
LOCAL_SAVE_AREA (2) = 0;     /* should precede restore */
```

or:

```
SAVE_AREA.STATUS (0) = 0;
```



NOTES:
S = Sign
BIT 0 OF EACH FIELD IS RIGHTMOST, LEAST SIGNIFICANT BIT OF CORRESPONDING REGISTER FIELD.
BIT 63 OF SIGNIFICAND IS INTEGER BIT (ASSUMED BINARY POINT IS IMMEDIATELY TO THE RIGHT).

121945-9

Figure G-4 Memory Layout of the REAL Save Area for the 8086 and the 80286 Microprocessors, and the 8087 and the 80287 Numeric Coprocessors

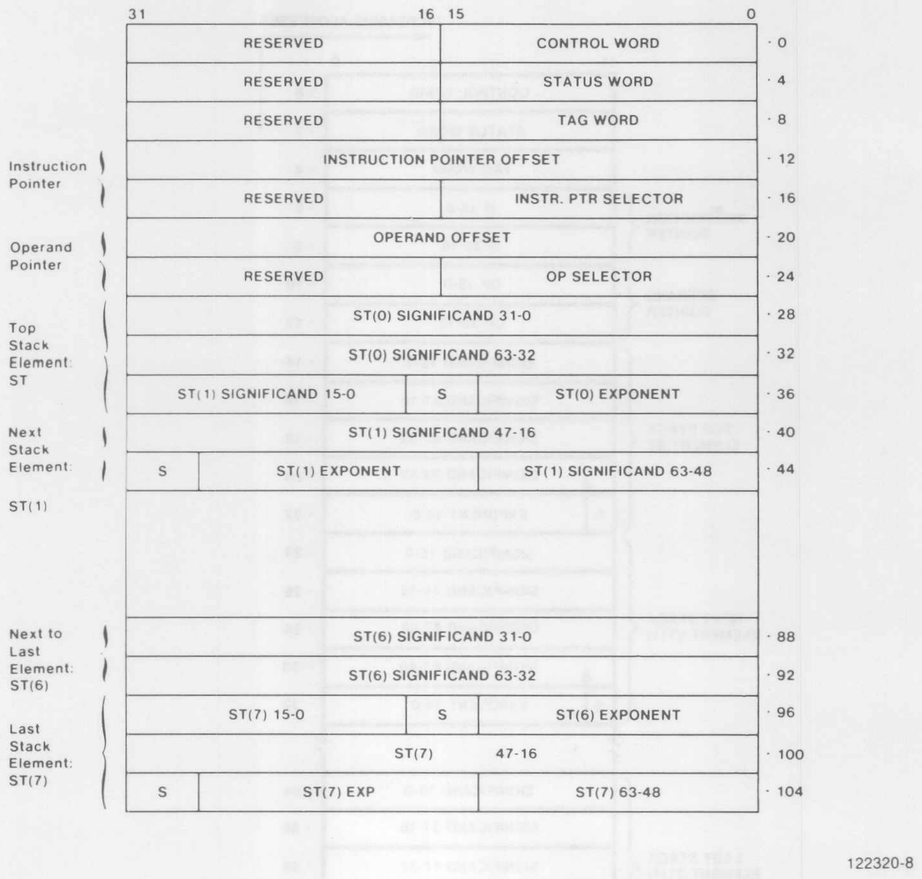


Figure G-5 Memory Layout of the REAL Save Area in Protected Mode for the 80386 Microprocessor

H RUN-TIME SUPPORT FOR PL/M APPLICATIONS

In addition to tools that support the software development process, Intel provides run-time support for application programs.

H.1 Numeric Coprocessor Support Libraries

Two libraries contained in the 8087 numeric coprocessor support library are of use to PL/M programmers: DCON87.LIB and CEL87.LIB. DCON87.LIB aids in number format translation (e.g., from binary to ASCII). CEL87.LIB is a common elementary function library that provides an assortment of functions involving floating-point numbers, such as rounding.

Three libraries contained in the 80287 numeric coprocessor support library are of use to PL/M programmers: CEL287.LIB, DC287.LIB, and EH287.LIB. CEL287.LIB includes common elementary functions (logarithmic, exponential, trigonometric, hyperbolic, etc.). DC287.LIB converts floating-point representations from ASCII decimal format to internal binary format, and vice versa. DC287.LIB is used by PL/M-286 and PL/M-386 programs to perform type conversions. EH287.LIB includes floating-point exception-handling procedures.

For additional information on the libraries contained with both the 8087 and the 80287 numeric coprocessors, see the numeric coprocessor reference manual.

H.2 PL/M Support Libraries

The PL/M libraries contain connection procedures and complex built-ins written in assembly language. The following library modules are provided in the PL/M libraries:

- PL/M-86 and PL/M-286 — LQ_DWORD_DIVIDE and
  LQ_DWORD_MULTIPLY
- PL/M-386 — INTERFACE286_FAR,
  INTERFACE286_NEAR,
  LQ_DWORD_DIVIDE,
  LQ_DWORD_MULTIPLY, MOVBIT,
  MOVRBIT, SCANBIT, and SCANRBIT.

Index

Tabs for
452161-001

INDEX